POWER-EFFICIENT DESIGN OF HIGH PERFORMANCE GOOGLENET-BASED CONVOLUTIONAL NEURAL NETWORKS HARDWARE ACCELERATOR

By

Ahmed Jamal Mohamed Abdel-Maksoud

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2021

POWER-EFFICIENT DESIGN OF HIGH PERFORMANCE GOOGLENET-BASED CONVOLUTIONAL NEURAL NETWORKS HARDWARE ACCELERATOR

By

Ahmed Jamal Mohamed Abdel-Maksoud

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

Under the Supervision of

Prof. Ahmed Hussien Mohamed

……..

Professor
Electronics and Communications Department
Faculty of Engineering, Cairo University

Dr. Hassan Mostafa Hassan

……..

Assistant Professor
Electronics and Communications Department
Faculty of Engineering, Cairo University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2021

POWER-EFFICIENT DESIGN OF HIGH PERFORMANCE GOOGLENET-BASED CONVOLUTIONAL NEURAL NETWORKS HARDWARE ACCELERATOR

By

Ahmed Jamal Mohamed Abdel-Maksoud

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

Approved by the
Examining Committee

_____
Prof. Ahmed Hussien Mohamed Khalil, Thesis Main Advisor

_____
Dr. Hassan Mostafa Hassan Mostafa, Advisor

_____
Prof. Mohamed Mahmoud Riad Alghoniemy, Internal Examiner

_____
Prof. Ahmed Hassan Kamel Madian, External Examiner
(Atomic Energy Authority & Nile University)

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2021

Engineer Name: Ahmed Gamal Mohamed Abdel-Maksoud
Date of Birth: 2 / 7 / 1995
Nationality: Egyptian
E-mail: ahmedjamal2035@yahoo.com
Phone: 01121040245
Address: Electronics and Communications Engineering Department, Cairo University, Giza 12613, Egypt

Registration Date: 1 / 3 / 2019
Awarding Date: …. /…. /2021
Degree: Master of Science
Department: Electronics and Communications engineering

Supervisors:

Prof. Dr. Ahmed Hussien Mohamed Khalil
Dr. Hassan Mostafa Hassan Mostafa

Examiners:

Prof. Ahmed Hussien Mohamed Khalil (Thesis main advisor)
Dr. Hassan Mostafa Hassan Mostafa (advisor)
Prof. Mohamed Mahmoud Riad Alghoniemy (Internal examiner)
Prof. Ahmed Hassan Kamel Madian (External examiner)
(Atomic Energy Authority & Nile University)

Title of Thesis:

Power-Efficient Design of High Performance GoogleNet-Based Convolutional Neural Networks Hardware Accelerator

Key Words:
CNN, Deep Learning, Image processing, artificial intelligence, hardware accelerators

Summary:

Convolutional neural networks (CNNs) have dominated image recognition and object detection models in the last few years. However, they require a huge cost of computation and a large memory size. This thesis presents a low-power convolutional neural networks hardware accelerator based on GoogLeNet. Several optimization and approximation techniques are applied to reduce the power consumption and memory size. Consequently, only FPGA BRAMs are used for weights storage without using offline DRAMs. In addition, the proposed hardware accelerator uses zero DSP units. The accelerator classifies 25.1 frames/sec with only 3.92W, which is more power-efficient than previous GoogLeNet FPGA implementations. The processor uses only 224 parallel elements (PEs) and achieves an average classification efficiency of 91%

Disclaimer

I hereby declare that this thesis is my own original work and that no part of it has been submitted for a degree qualification at any other university or institute.

I further declare that I have appropriately acknowledged all sources used and have cited them in the references section.

Name: Ahmed Jamal Mohamed Date: .. /.. /2021

Signature: Ahmed Jamal Mohamed

Dedication

To my parents, my brother, and my family whose unbounded support and love have brought me this far.

Acknowledgments

I would like to offer my sincere thanks to my supervisors Dr. Ahmed Hussien and Dr. Hassan Mostafa. I would express my deepest gratitude to Dr. Hassan for his efforts throughout this work for precious supervision, continuous encouragement, and active help.

I want to thank all those, who helped me by their knowledge and experience. I will always appreciate their efforts.

My sincere gratitude and love to my family for their encouragement and help during whole masters journey.

Table of Contents

List of Tables

List of Figures

Nomenclature

CNN	Convolutional Neural Network
AI	Artificial Intelligence
ANN	Artificial Neural Networks
RTL	Register Transfer Logic
DSP	Digital Signal Processing
DRAM	Double Random Access Memory
BRAM	Block Random Access Memory
PE	Parallel Element
ML	Machine Learning
DL	Deep Learning
GPU	Graphical Procession Units
ASIC	Application-Specific Integrated Circuits
FPGA	Field-Programmable Gate Arrays
IP	Intellectual Property
RISC	Reduced Instruction Set Computing
CISC	Complex Instruction Set Computing
MAC	Multiply and Accumulate
HLS	High-Level Synthesis
OpenCL	Open Computing Language.
INQ	Incremental Network Quantization
FM	Feature Map
IFMAP	Input Feature Map
OFMAP	Output Feature Map
CU	Control Unit
PU	Processing Unit
WCU	Weight Control Unit
FC	Fully Connected
LRN	Local Response Normalization
BN	Batch Normalization
SAIF	Switching Activity Interchange Format

Abstract

Convolutional neural networks (CNNs) have dominated image recognition and object detection models in the last few years. They can achieve the highest accuracies with several applications such as automotive and biomedical applications. CNNs are usually implemented by using Graphical Processing Units (GPUs) or generic processors. Although the GPUs are capable of performing the complex computations needed by the CNNs, their power consumption is huge compared to generic processors. Moreover, current generic processors are unable to cope up with the growing CNNs demand for computation performance. Therefore, hardware accelerators are the best choice to provide the required computation performance needed by the CNNs as well as affordable power consumption. Several techniques are adopted in hardware accelerators such as pruning and quantization.

In this thesis, a power-efficient convolutional neural networks hardware accelerator is proposed based on GoogLeNet CNN. Weights pruning and quantization are applied, which reduces the memory size by 57.6x. Consequently, only FPGA BRAMs are used for weights and activations storage without using offline DRAMs. In addition, the proposed hardware accelerator uses zero DSP units as it replaces all multiplications by shifting operations. The accelerator is developed based on a time-sharing/pipelined architecture, which processes the CNN model layer by layer. In addition, there are some dedicated units such as maxpooling and average pooling units. The architecture proposes a new data fetching mechanism that increases data reuse. Moreover, it uses only 224 parallel elements (PEs). All the proposed accelerator units are implemented in native RTL (Register Transfer Logic), and several optimization techniques are applied to reduce the power consumption. The accelerator classifies 25.1 frames/sec with 3.92W only, which is more power-efficient than previous GoogLeNet FPGA implementations. In addition, it achieves top-5 average classification efficiency of 91%, which is significantly higher than comparable architectures. Furthermore, this accelerator overcomes the popular CPUs such as Intel Core-i7 and GPUs such as GTX 1080Ti in terms of the number of frames processed per Watt. The normalized power efficiency is 6.4 frames/Watt for the proposed accelerator, 0.81 frames/Watt for NVidia GPU, and 0.128 frames/Watt for Intel Core-i7.

Chapter 1 : Introduction

1.1. Motivation

Deep learning has been employed in a lot of domains during the last decade, such as image classification [1-2], object recognition and detection [3-5], object detection [6-7], audio recognition [8], and self-driving cars [9-10]. CNNs are used widely as they achieve challenging accuracies, and their models are easily applied to new applications. CNNs are one of the common deep learning algorithms mainly used for image and video classification and detection [11]. CNNs require large amounts of memory storage as there are millions of parameters in every CNN model. Moreover, CNNs are computationally intensive as they require billions of operations per image. The high computational complexity combined with inherent parallelism in these models makes them an excellent target for custom accelerators.

Although the CNNs have dominated the image classification and detection algorithms, there are two main challenges regarding their implementations [12]. The first challenge is the cost of computation, as their architecture consists of many convolutional layers, which are multiplication-hungry layers. The second challenge is the memory bandwidths, in which the memory fetching speeds are much lower than the processing speeds. These two challenges have raised the need to develop custom architectures to accelerate the CNN computations while keeping the power consumption at affordable rates for limited energy embedded applications. However, the variations of network architectures and data fetching patterns make it difficult to adopt one architecture for all CNNs. As a result, custom designs are the dominant approach for these networks to get the best performance across all performance metrics.

During the rising of deep learning (DL) and machine learning (ML) algorithms, two main categories of processors are used. The first platform is the Central Processing Units (CPUs), which are not efficient for DL and ML algorithms as these algorithms require high parallelism and a lot of DSP units to finish their processing rapidly. The second platform is the Graphical Procession Units (GPUs), which are capable of processing millions of pixels within a part of the second. Correspondingly, the GPUs are the most suitable platforms due to their high parallelism. Consequently, they have been used widely for both training and inference [13].

When it comes to hardware accelerators, FPGAs get a critical mission to provide high-performance – low power processing units [14-15]. FPGAs stand for field-programmable gate arrays (FPGAs) that provide low power consumption, high parallelism, optimized hardware, and real-time computation capabilities. Moreover, FPGAs have the advantages of short time-to-market, reconfigurability, and reusable IP (Intellectual Property) options. There is another choice for designers, which is ASIC chips. ASIC is application-specific integrated circuits that provide the lowest power consumption and highest clock speeds, but it has a long time to market and high initial fabrication costs. These properties make it suitable for mass production, such as NVidia accelerators and google TPUs or data centers, such as google cloud or amazon AWS.

As artificial intelligence (AI) is emerging increasingly in a lot of applications, the demand for hardware accelerators is increased. Recently, a lot of research is done to develop high-performance hardware accelerators for data centers, smartphones, and IoT

devices. Accelerator specifications are set based on the target application, power consumption budget, and acceleration rate. AI accelerators need more specialized architectures and should be suitable and optimized for the target algorithm, in contrast to common architectures, such as RISC (Reduced instruction set computing) and CISC (Complex instruction set computing) architectures. This approach is becoming more common in industrial and research applications, specially inference processors [16].

For many years, it is well-known that the depth of the network should be increased to get higher accuracies, especially the number of convolution layers. This has been a common direction till year 2014 when Szegedy proposed a new CNN network called GoogLeNet with the concept of inception module [17]. In this network, the depth and width of the network have been increased, but the computational budget has been kept constant by using the network-in-network concept. This concept uses additional 1x1 convolutional layers to remove the network bottlenecks to help in dimension reduction as shown in Figure 1.1. GoogLeNet overcomes AlexNet [1] and VGG [2] networks by getting the highest accuracy with fewer weights. As AlexNet uses 60 million weights to get 84.7% top-5 accuracy, and VGG-16 uses 138 million weights to get 92.7% top-5 accuracy. GoogLeNet uses only 6.9 million weights to get 93.4% top-5 accuracy. Despite all these advantages, GoogLeNet architecture is more complex than other CNN networks due to activations' data dependency and complex connections between inception layers. This makes it usually challenging for hardware accelerators designers.

1.2. The Proposed Work

This thesis will explore the lowest power consumption techniques for CNN hardware accelerators to make full use of it through the design. The key points for this research will be as follows:

- It is a dedicated hardware accelerator that is designed for CNNs.
- It is an inference processor that uses pre-trained weights.
- The main design purpose will be achieving the lowest power consumption.
- It is designed on FPGA for fast prototyping and reconfigurability.
- It is implemented using native RTL (Verilog).
- It is specially designed for GoogLeNet CNN for best performance.
- Providing a suitable/optimized architecture to meet the computation requirements.

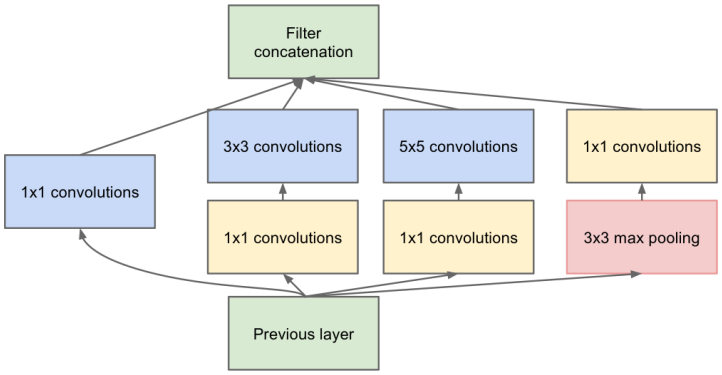


Figure 1.1: Inception module with dimension reduction [17]

The main features of the proposed accelerator in this work are briefly highlighted as follows:

i. The accelerator achieves 25.1 fps for GoogLeNet classification with 3.92W, which provides more power-efficiency than previous FPGA implementations for GoogLeNet.
ii. The accelerator achieves an order of magnitude performance improvement over Intel Core-i7 and NVidia GTX 1080Ti.
iii. Weights pruning and quantization are used to cut down the memory usage by 57.6x. As a result, only FPGA BRAMs are used for weights and activations storage without using offline DRAMs.
iv. It uses zero DSP units by converting all multiplications into shifting operations.
v. This accelerator is developed based on time-sharing/pipelined architecture that processes the CNN model layer by layer.
vi. This accelerator proposes a new data handling mechanism that leads to high data reuse and low power consumption.
vii. The proposed accelerator uses simple distributed control units, which can be reconfigured to other CNNs such as VGG.
viii. The proposed accelerator uses only 224 simple parallel elements (PEs).
ix. The design achieves top-5 classification accuracy of 91%, which is significantly higher than comparable architectures.

1.3. Thesis Organization

This thesis is organized as follows, Chapter 1 presents the introduction and motivation for this work in addition to the proposed work main features and thesis organization. Chapter 2 gives a brief background about neural networks and CNNs. Moreover, a literature review is made, to sum up all related work and show the areas that can be developed. Chapter 3 presents the applied memory compression model using both weights pruning and quantization. The results are used lately for improving the architectural design and optimizing the power consumption and the utilized area. Chapter 4 investigates the proposed architecture in details. Moreover, design and implementation for each block are discussed. In Chapter 5, the experimental results and discussions for the implemented design are presented. Different analyses are held to make sure of the accelerator performance. Finally, Conclusion is presented to conclude the contributions and show the future work.

Chapter 2 : Background and Literature Review

In this chapter, the background necessary for the next chapters are presented. Firstly, neural networks are introduced. Secondly, the main layers of CNNs in addition to their training and inference information are presented. Finally, a literature review is presented about algorithm-level optimization techniques, hardware design methods, and popular CNN hardware accelerators implementations.

2.1. Background

2.1.1. Neural Networks Overview

Artificial intelligence has been evolved through the years and split into several branches such as robotics, NLP, machine learning, and neural networks [48]. Neural networks usually consist of multiple layers. When the number of layers is increased, the depth of the network increases, which is stated as Deep Neural Network (DNN). A neural network basically consists of a system of neurons with an artificial nature, where artificial neurons are known as perceptrons. A complete perceptron model represents a complete neural network. Neural network algorithms usually find elemental relationships across the input data, and find the best model or system to represent it.

A simple perceptron takes several inputs and generates a single output as shown in Figure 2.1. It is developed by Frank Rosenblatt in the 1950s and 1960s [43]. The perceptron is mathematically represented as an activation function. It is a non-linear function that allows output triggering with small changes in the weights or bias after passing the threshold. It is stated as follows:

$$y = f(W.X + b) = f(\sum_j w_j x_j + b) \tag{2.1}$$

where the y is the perceptron output, which is calculated by the dot product, followed by the bias addition. The vector W is the neuron weights, and vector X is the input. Finally, b is the bias.

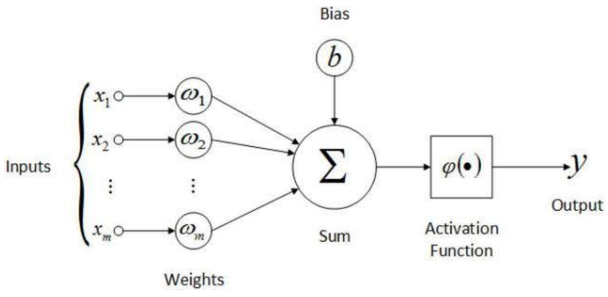


Figure 2.1: A simple perceptron

A conventional neural network is constructed with thousands or millions of neurons that are usually organized into multiple layers. The first layer of a neural network is the input layer, which is followed by one or multiple hidden layers. After the last hidden layer, the classification is made by the output layer.

There are many different activation functions, but ReLU, Tanh, and sigmoid are the commonly used ones [44]. Each activation curve is shown in Figure 2.2. They are deployed in the network based on their functionality. They may be used for adding non-linearity to the layers or transform the classification results into probabilistic values. Every function is illustrated briefly as follows:

i. ReLU function

ReLU simply suppresses any negative value to zero. It is used in hidden layers to add non-linearity to the layers' output. The mathematical representation is written as follows:

$$\mathrm{ReLU}(z) = \max(0, z) \tag{2.2}$$

where $ReLU(z)$ is the output and z is the input.

ii. Sigmoid function

It is used in output layer for classification as it gives a weighted output. The mathematical representation is written as follows:

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.4}$$

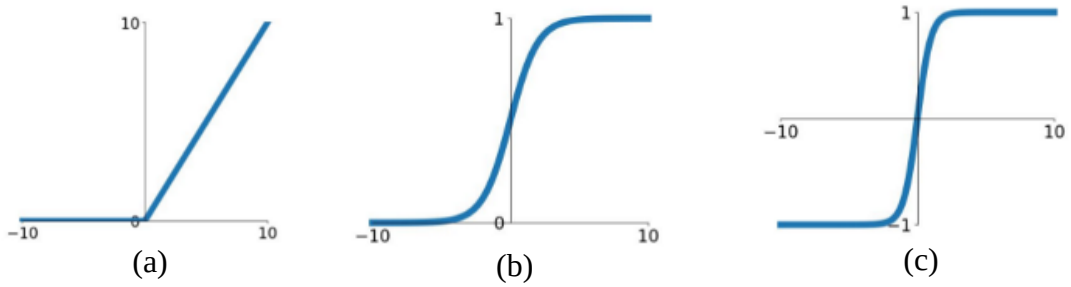
where $\sigma(z)$ is the output and z is the input.

iii. Tanh function

It is used in hidden layers to add non-linearity to the layers' output. The mathematical representation is written as follows:

$$tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \tag{2.3}$$

where $tanch(z)$ is the output and z is the input.



(a) (b) (c)

Figure 2.2: Activation functions, (a) ReLU, (b) Sigmoid, and (c) Tanh functions

2.1.2. Multilayer Perceptron

The multi-layer perceptron (MLP) is one of the Feed-Forward neural networks, where the propagation of data only goes from the input to the output layers during calculations [45]. There is another type of neural networks which is the Recurrent Neural Networks (RNN). RNNs in contrast, have a feedback connections from forward layers to previous layers [46]. MLP is the most basic configuration of feed-forward neural networks and is commonly referred as fully connected layers. MLP usually consists of one or more hidden layers of perceptrons where the network's perceptrons are connected with each other. When the number of hidden layers increases, the network deals with more complex problems. On the other hand, the number of network parameters jumps rapidly which increases the cost of training and inference. A simple MLP is as shown in Figure 2.3 which has one input layer, one hidden layer, and one output layer.

2.1.3. Convolutional Neural Networks

Convolutional Neural Network (CNN) is one of the feed-forward classes of neural networks, and is most mainly used for vision tasks [47]. CNNs, in contrast to MLPs, do not require every neuron in the input layer to receive information from every pixel of the visual field, which in turn simplifies the network complexity and connections.

As all artificial neural network models are inspired by the human brain. The brain analogy usually identifies an object or a photo by describing the distinguishing features such as edges, color, and main shapes. In this way, it works efficiently without requiring the position and color or every pixel.

In CNNs, kernels or filters are responsible for feature extraction through an operation that is well-known as 2D convolution. The convolution kernel slides across all input feature map while calculating the cross-correlation between the input feature map and applied kernel. The output is a scalar value that corresponds to how similar the input is to the kernel as shown in Figure 2.4. The convolution kernels are kept constant throughout the frame traversal and every CNN layer distinguishes multiple features of

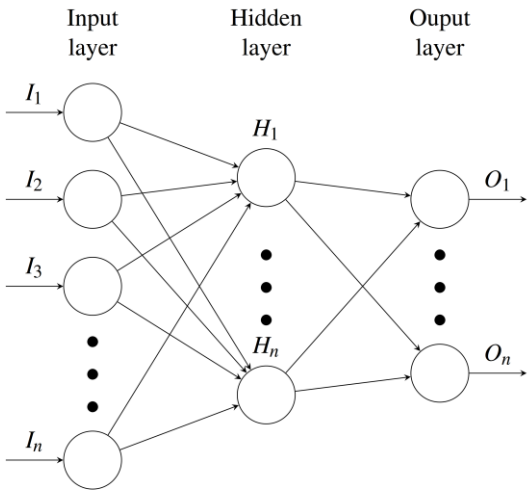


Figure 2.3: Basic MLP structure

the frame. Usually, the input images are not represented as two-dimensional arrays, but with multichannel inputs such as RGB images. This forms three-dimensional arrays with 3D-convolution operations.

CNN basically consists of convolutional layers, fully connected layers, and pooling layers. These layers are stacked several times to form a CNN [47]. The deeper every CNN becomes, the higher accuracy it provides. However, this approach is evolved with time to add different techniques rather than increasing the depth of the network. Figure 2.5 shows a simple CNN that consists of 3 convolution layers and 3 fully connected layers in addition to Maxpooling layers. The main layers for a conventional CNN will be discussed briefly in the next section.

2.1.3.1. Convolution layers

Convolution layers are one of the main building blocks of CNNs as they are responsible for extracting local features from input feature maps [47]. They require a lot of computations due to convolving nature and the size of convolutions. The mathematical equation for 2D convolution operation is stated as follows:

$$y[m, n] += \text{bias} + \sum_{k=1}^{K-1} \sum_{l=1}^{L-1} w[k, l]\, x[m + k, n + 1] \tag{2.5}$$

where y[m,n] is the output matrix map of the same dimensions as the input map x. m and n are the coordinates of the pixel of the interest region. Finally, L and K denote the kernel dimensions

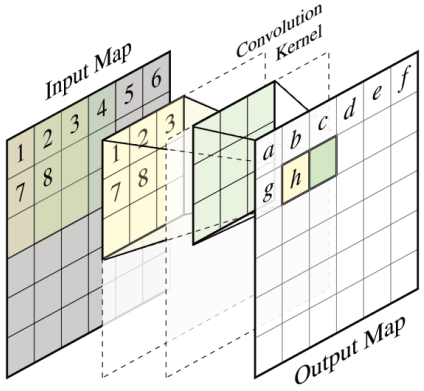


Figure 2.4: 2D-Convolution with sliding window kernel

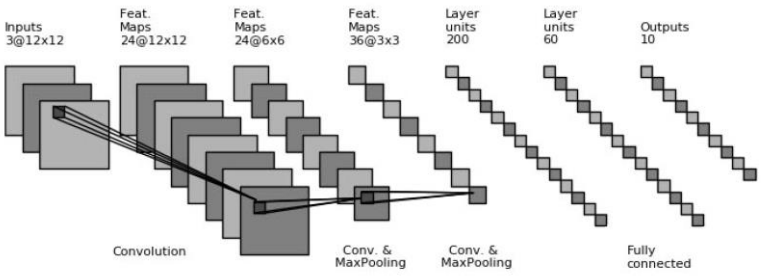


Figure 2.5: A simple CNN

Convolution layer dimensions are controlled by 4 hyper-parameters which controls the convolution input/output feature map sizes:

- The number of filters K.
- F is the width and height of filters.
- The stride S, the kernel sliding step.
- Finally, P is the amount of zero padding pixels.

Where the size of the output feature maps is calculated as follows:

$$OFMAP_{dimen} = \frac{IFMAP_{dimen} - F + 2P}{S} + 1 \quad (2.6)$$

2.1.3.2. Fully connected layers

Fully connected layers are used at the end of each CNN model to compute class scores as shown in Figure 2.6 example. It is like MLP which has one-to-one connections to all previous layer activations. There may be multiple fully connected layers in every exact CNN such as AlexNet to reduce the input size gradually and avoid having a huge full-connectivity layer at the end. Every CNN ends up with a fully connected layer with size equals to the number of trained classes.

2.1.3.3. Pooling layers

There are two popular pooling layers. Firstly, Maxpooling layers are usually paired with the convolution layers. Maxpooling works on each feature map separately by taking the max value of the applied subregion. The second type is the average pooling layer, which is applied by taking the average value of the applied subregion. Although it has been the most popular historically, it is less commonly used nowadays.

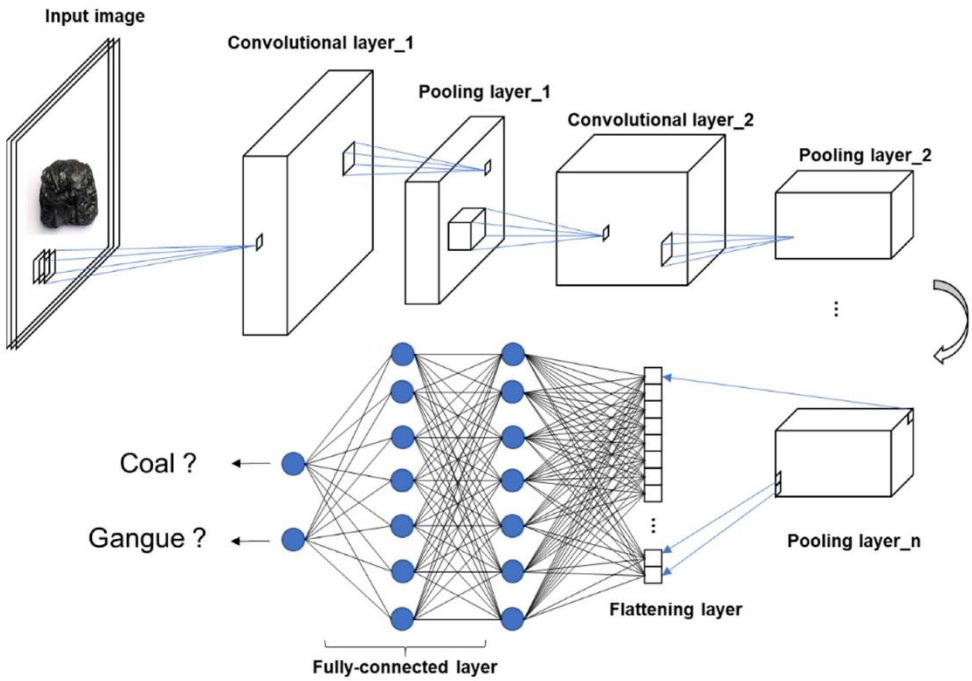


Figure 2.6: Fully Connected layer in CNN models

Pooling layers are used to shrink the size of the input feature maps to reduce the number of parameters in the CNN model, hence reducing the amount of computation needed. On the other hand, they overcome the overfitting problem. Overfitting is a problem that occurs when a network predicts and classifies the training data well, but fails to deal with testing data. The pooling operation is performed on each feature map, reducing the size of each feature map without removing any of them. The pooling layer requires two hyper-parameters, F and S where the size of the output feature maps is calculated directly using (2.7):

- Subregion width or height, F.
- The stride S.

$$OFMAP_{dimen} = \frac{IFMAP_{dimen} - F}{S} + 1 \qquad (2.7)$$

Figure 2.7 shows a simple example for Maxpooling operation with input feature map size 4x4, subregion F=2, and stride 2. The output feature map size will be 2x2 by following the equation of (2.7)

2.1.3.4. Other layers

After presenting the most popular layers, there are other types of layers that have importance in CNN calculations. Firstly, the normalization layers that are used to normalize the output at some parts of the network such as local response normalization (LRN) or batch normalization (BN) which is applied after each convolution layer [41]. Although BN has an additional cost of having learnable parameters and extra computations, it is preferred nowadays over LRN. However, LRN does not has any additional learnable parameters.

On the other hand, there are some activation layers like ReLU activation, which is commonly used to add some non-linearity to network feature maps. Also, there is the dropout layer which is useful while training to overcome overfitting by randomly dropping some inputs. Their functionality is disabled while inference.

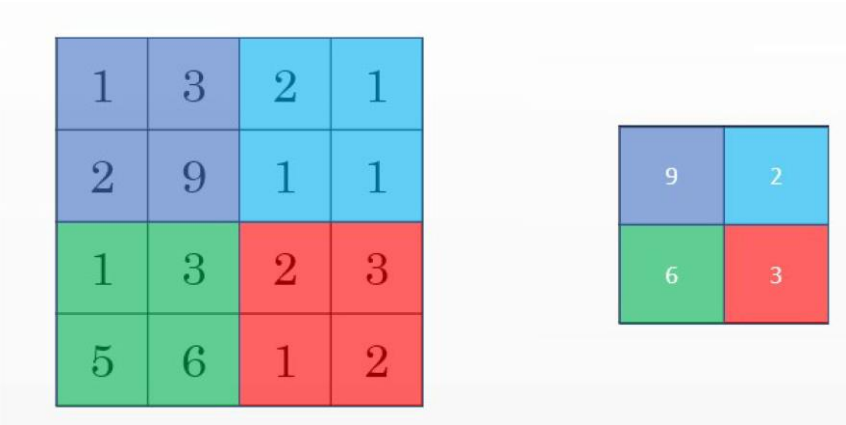


Figure 2.7: Maxpooling example with F=2 and S=2

2.1.4. Popular CNN Models

In this section, a careful analysis for popular CNNs is presented. Then, the analysis compares GoogLeNet CNN with other popular CNN models.

- LeNet

LeNet CNN is one of the most basic CNNs. It's consists of two convolution layers, two average pooling layers, two fully connected layers, and one Softmax layer as shown in Figure 2.8. It was originally developed in 1998 to identify the hand written digits with 32 x 32 x 1 input grayscale image size. It has 60 K parameters. Moreover, LeNet uses Sigmoid and Tanh activation functions

- AlexNet

AlexNet was developed to classify 1000 classes of ImageNet Dataset [1]. It is consists of 60 Million parameters with deeper layers than LeNet, so it required multiple GPUs for training. The input image size is increased to 227 x 227 x 3 which is RGB input. Also, the local response normalization layer is firstly introduced, and maxpooling is used instead of average pooling. The CNN structure is shown in Figure 2.9 with a layer arrangement with (Conv-1, Maxpool-1, Conv-2, Maxpool-2, Conv-3, Conv-4, Conv-5, Maxpool-3, FC-1, FC-2, Softmax). AlexNet won the ImageNet challenge with 83.6% top-5 classification accuracy.

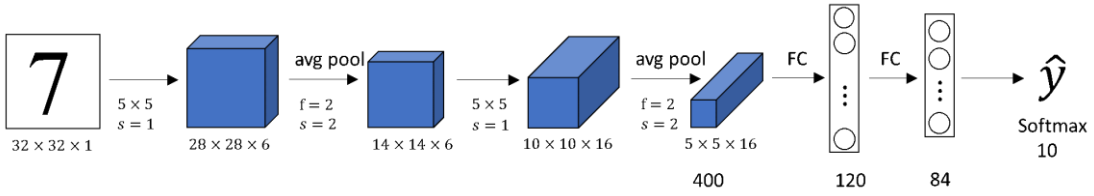


Figure 2.8: LeNet CNN

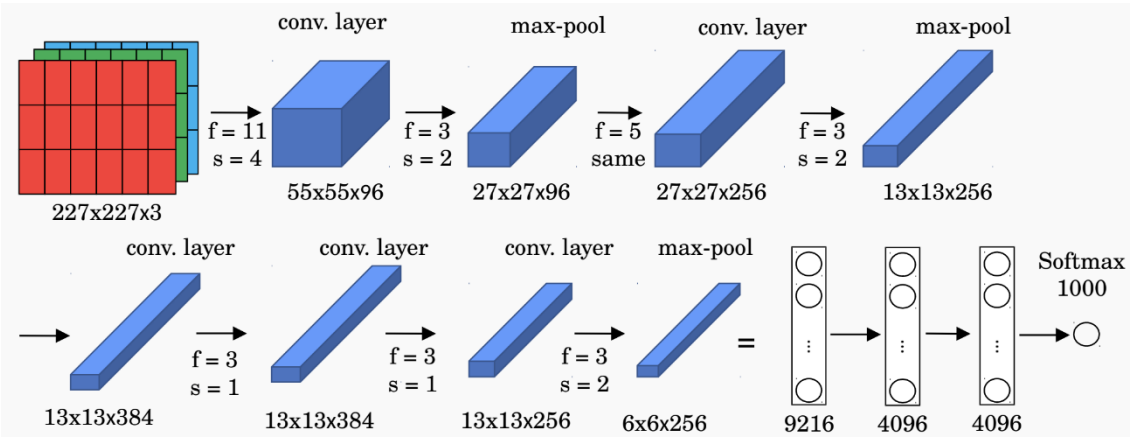


Figure 2.9: AlexNet CNN

- VGG-16

VGG-16 CNN is a modification for AlexNet which was developed for ImageNet challenge [2]. The size of the model is large with 138 Million parameters, so it required huge resources for the training process. The input image size is set as 224 x 224 x 3 which is RGB input. The CNN structure is shown in Figure 2.10. Convolution filters are increased from 64 to 128 to 256 to 512, and Maxpooling layer are responsible for shrinking the input size. VGG-16 won the second place in ImageNet challenge with 92.7% top-5 classification accuracy in 2014.

- Inception V3

Inception V3 is the third version for GoogLeNet, which supports the same idea of inception network using 23.6 Million parameters [54]. The number of parameters for Inception V3 is more than GoogLeNet (6.9M), but less than the number of parameters for AlexNet (60M). Inception V3 achieves top-5 classification accuracy with 96.5%, which won the ImageNet challenge in 2015. Figure 2.11 shows the structure for Inception V3.

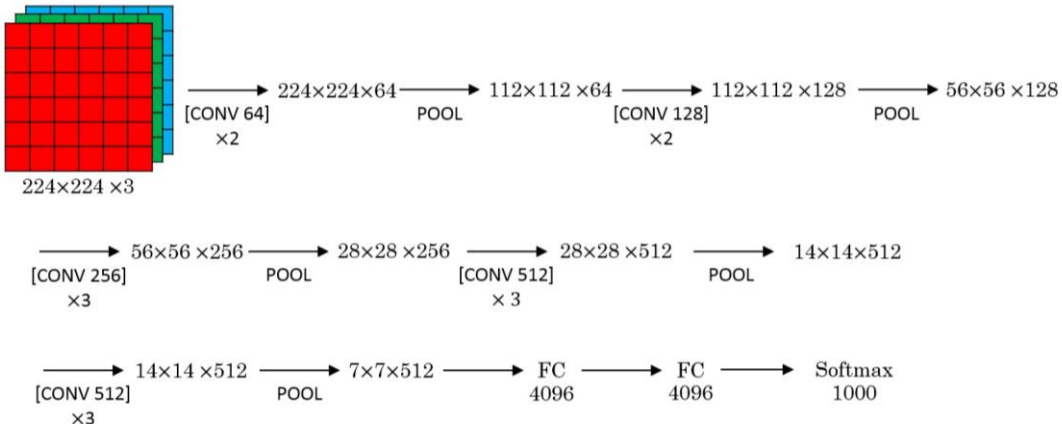


Figure 2.10: VGG-16 CNN

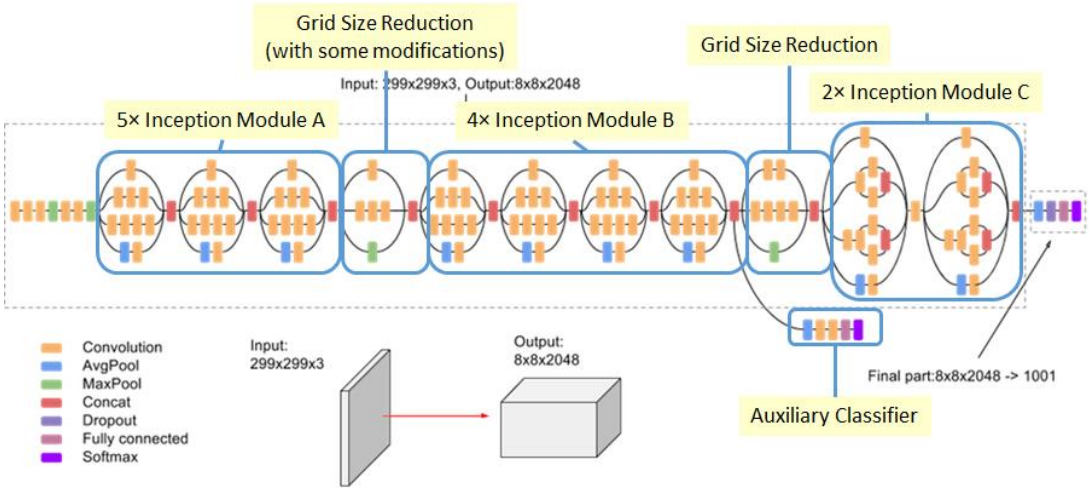


Figure 2.11: Inception V3 CNN

- GoogLeNet

GoogLeNet CNN achieves a top-5 classification accuracy with 93.4% using ~6.9 Million parameters only [17]. The number of parameters is cut down after deploying the concept of network-in-network module. This module uses 1x1 convolution layer before 3x3 convolution and 5x5 convolution layers to shrink the dimension. It's one of the most remarkable CNN provided in the literature.

- ResNet

ResNet CNN stands for residual network which consists of a group of residual blocks. The main idea is to add a skip connections before the second activation [55]. ResNet is the first CNN to allow the training of very deep networks even if with more than 100 layers. ResNet-34 achieves top-5 classification accuracy with 94.4% using 20.5 Million parameters. There are many variations for ResNet such as ResNet-50 and ResNet-152. Figure 2.13 shows an example for ResNet-34 CNN.

- SqueezeNet

SqueezeNet CNN is usually compared with AlexNet as it achieves the same classification accuracy with 50x fewer weights [56]. SqueezeNet consists mainly of fire modules with 3x3 and 1x1 convolution kernels. The 1x1 filters are used to reduce the input feature map size before 3x3 filters. SqueezeNet starts with a convolution layer, followed by 8 fire modules, and ends with a final convolution layer as shown in Figure 2.12.

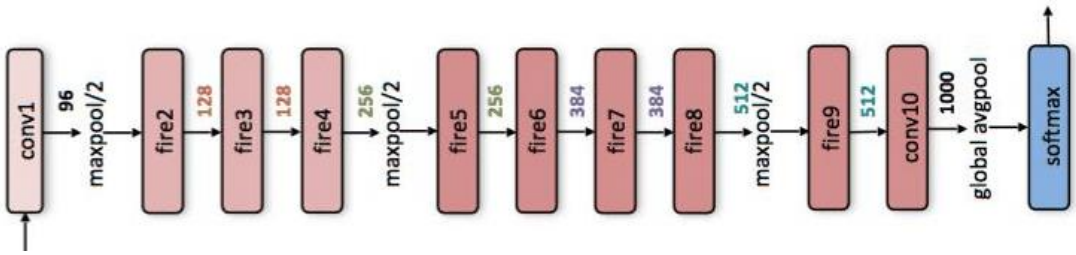


Figure 2.12: SqueezeNet CNN

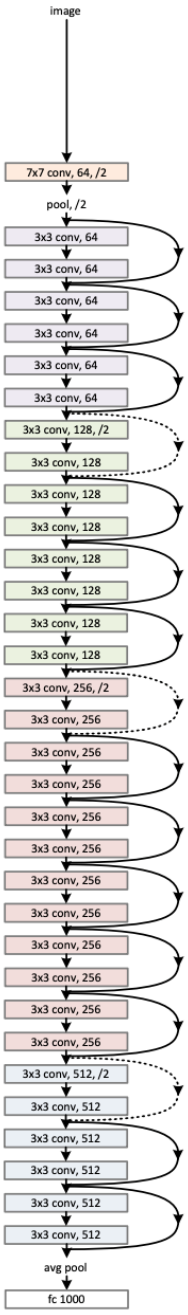


Figure 2.13: ResNet-34 CNN

In Table 2.1, a summarized comparison is made between popular CNN networks. VGG-16 has a Top-5 accuracy with 92.7%, but it requires 138 Million weights per frame. This requires a huge memory size, which in turn will increase the computation load and memory access. On the other hand, Inception V3 and ResNet-50 get higher accuracy with 96.5% and 96.4%, respectively. They get an approximate accuracy improvement of 3% more than GoogLeNet, but they require memory storage 3.5x times more than GoogLeNet. SqueezeNet gets the AlexNet accuracy with 50x fewer parameters, but top-5 accuracy with 83.6% is low compared with other CNNs accuracies. Accordingly, it is clear that GoogLeNet achieves the best accuracy while keeping the number of weights in an acceptable count.

Table 2.1: Popular CNNs

CNN	Year	Top-5 accuracy (%)	Number of weights (Millions)
AlexNet	2012	83.6	60 M
VGG-16	2014	92.7	138 M
GoogLeNet	2014	93.4	6.9 M
ResNet-34	2015	94.4	21.5 M
Inception V3	2015	96.5	23.6 M
SqueezeNet	2017	83.6	1.2 M

Most of the hardware accelerators in the literature proposes high throughput and reasonable power consumption on feed-forward CNN networks such as LeNet, AlexNet, and VGG [37-38]. These processors fail to process the inception network well, and the obtained speed is degraded, since the structure of the inception module increases the depth of the layers horizontally and vertically while keeping computational cost by adding a 1x1 convolution layer as a bottleneck. Although this improves the accuracy, it increases system complexity. The challenge is to build a hardware accelerator based on GoogLeNet model, and design it carefully to make full use of every feature of it.

2.1.5. Neural Networks Training

The process of training a network is responsible for evaluating the optimum value of all learnable parameters such as weights and biases of the network. As the training of the networks is outside the scope of this work, only a quick overview of network training will be explained.

After constructing the model of the neural network, network training comes to adjust the parameter values for all weights in the network. The training process is controlled via hyper-parameters and a training dataset. The desired goal of this process is to be able to classify similar classes during inference. Firstly, a cost function is introduced. The goal is to minimize this function to the least value. Cost function represents the squared difference between the computed output value and the desired one which is stated as follows:

$$C(w,b) = \frac{1}{m}\sum_{i=1}^{m}\left|\left|y-\hat{y}\right|\right|^2 \tag{2.8}$$

Gradient decent algorithm is used to minimize the cost function. Gradient descent updates the values of the weights and biases with small steps in the direction of the negative gradient. This update for each parameter is given by:

$$w'_k = w_k - \eta\frac{C(w,b)}{w_k} \tag{2.9}$$

$$b'_k = b_k - \eta\frac{C(w,b)}{b_k} \tag{2.10}$$

Where η is the learning rate or correction step, which is one of the training hyper-parameters. If the learning rate is too small it leads to a very slow convergence being stuck in local minima. On the other hand, if the learning rate is too high it leads to a non-convergence system, so that it requires a careful selection.

Unfortunately, the gradient descent algorithm cannot be really applied for training neural networks as it requires complex computations and all training dataset is involved in each step, which is impossible to be applied. There is another algorithm called Mini-batch Gradient Descent. In Mini-Batch Gradient Descent, the derivate is approximated on a small mini-batch of the dataset, and is used to update the weights. Mini-batch is not guaranteed to reach an optimal solution. But, if a small learning rate is chosen with gradient descent, the loss is guaranteed to decrease every iteration.

On the other hand, computing exact derivatives for millions of parameters is hard for a typical neural network model. Another way to calculate the derivatives is called back-propagation, which gives a good balance between the results and computations. The back-propagation method firstly computes the forward-propagation path in order to compute the output. Secondly, the error values on the outputs are propagated backward through the network, which is used to calculate the gradients and update all network parameters [46].

In brief, the training process nowadays uses both concepts of forward-propagation and back-propagation. The objective is not to have the minimum difference between the current weight value and the desired weight, but reaching to minimum error (Loss) between the classification of the training data and the prediction are made by the neural network.

Training of artificial neural networks is simplified in steps as follows:

1. Start by random weight initialization.
2. Split the dataset into batches with the same batch size.
3. Train the network with the batches, one by one.
4. Perform the forward-propagation to get the output with the values of the current weight.
5. Compare the calculated output to the expected output and compute the loss.
6. Update the weights using backward-propagation with a decrement or increment learning rate.
7. Repeat the process with other batches till finishing the training batches.

2.1.6. Neural Networks Inference

After performing network training, the model weights are saved to be used while applying the model in its application. The inference phase is simpler than the training as it just computes the forward-propagation path of the network, and evaluates the output without going backward for back-propagation and so on. Training accuracies are usually more than inference accuracies because the model fits the training data well and all network weights are adjusted for it. However, researchers seek to keep the inference accuracy so close to training accuracy.

2.2. Algorithm-Level Optimization Techniques

In this section, a brief review about popular techniques for efficient hardware accelerator design using algorithm-level optimizations. These methods are addressed to select the suitable ones for the proposed hardware accelerator. The studied techniques are Weights pruning, quantization, weights sharing, Huffman coding, winograd transformation, binary/ternary nets, and low-rank approximation.

1. Weights Pruning

Deep neural networks models consist of millions of parameters. Many of these parameters are not important, and removing them can reduce the weights memory storage greatly. This is called the weights pruning operation, which eliminate the unnecessary connections. Figure 2.14 shows a simple example for a neural network before and after pruning. The unnecessary synapses are pruned away which simplifies the overall connections greatly. Eliminating these connections degrades the model accuracy as the model connections are mutually dependent. Correspondingly, retraining the remaining connections is necessary to recover the accuracy loss [57]. Figure 2.15 shows the accuracy loss versus the percentage of pruned parameters for pretrained, pruned, and pruned/retrained models. It's depicted from the figure that the accuracy starts to decrease by increasing the percentage of pruned-away parameters. By comparing the pruned-only model with pruned/retrained model, it's clear that retraining keeps the accuracy loss tends to zero while saving 80% of the model weights. The pruning percentage decreases by increasing the model's complexity, but this provides a good example about the effect of retraining after weights pruning on the model size. It's worthy to mention that iterative retraining improves the accuracy more than one-stage retraining.

Weights pruning makes a large change in the weights distribution. Basically, the pruning suppresses the weights that tend to zero a shown in Figure 2.16 (b). Retraining affect is shown in Figure 2.16 (c), which retrains the model weights to get a continuous distribution and recover the accuracy loss.

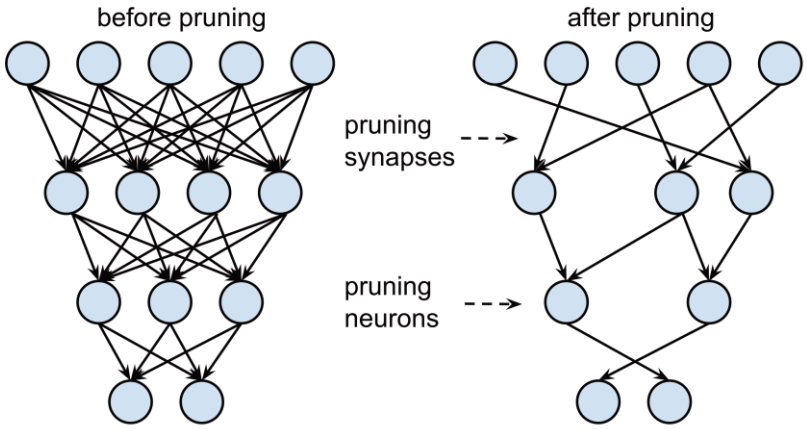


Figure 2.14: Simple neural network before and after pruning

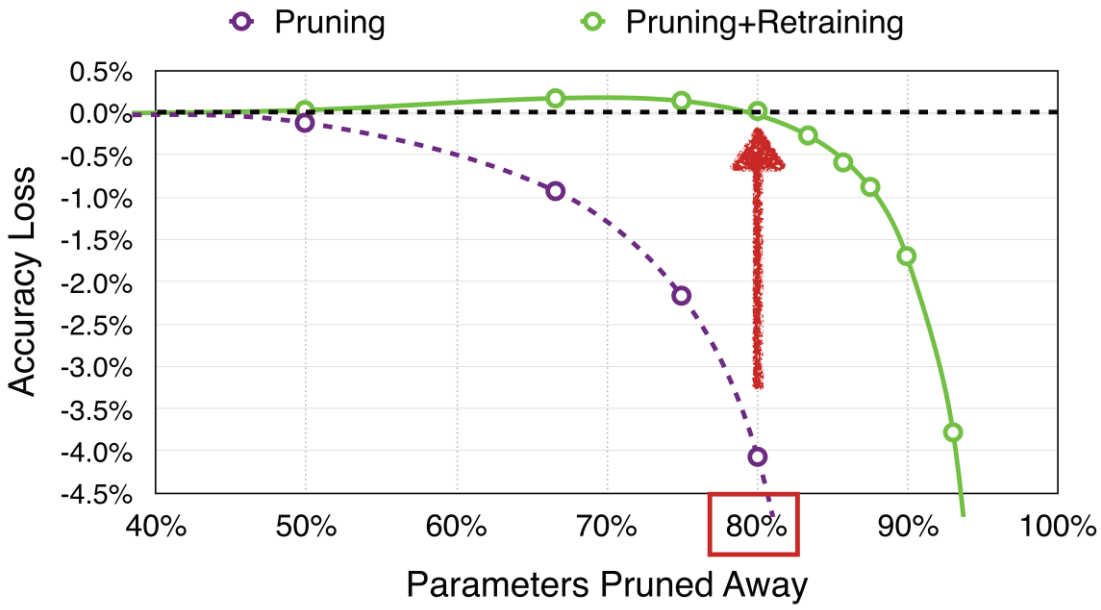


Figure 2.15: Accuracy loss versus the number of pruned parameters for pretrained, pruned, and pruned+retrained models.

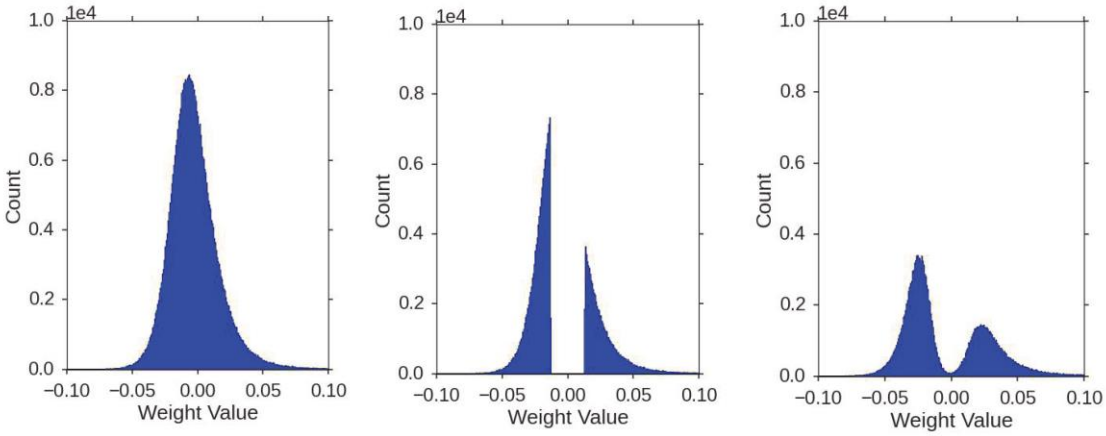


Figure 2.16: Weights distribution changes with pruning. (a) Before pruning, (b) after pruning, and (c) After retraining.

2. Quantization

Weights of deep learning models are usually represented with 32-bit precision, which allocates a large size of memory. Representing these weights in a smaller precision reduces the model size significantly. Moreover, quantizing the weights reduces the accuracy directly, and accordingly, retraining is necessary for quantization as the same as weights pruning. After applying weights pruning model, weights quantization is used to shrink the precision from 32-bit to 4-bit. Iterative quantization is preferred more than quantizing all the weights in one shot as it reduces the accuracy loss by partitioning the weights iteratively. Figure 2.17 shows the weights distribution

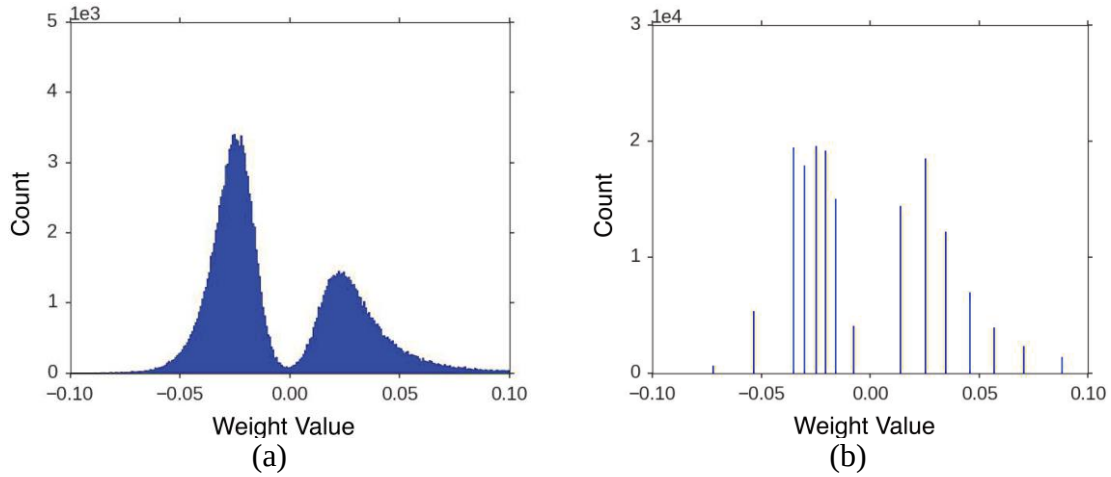


Figure 2.17: Weights distribution changes with quantization. (a) Before trained quantization, and (b) After trained quantization.

before and after trained quantization. It's clear that the quantization makes the weights distribution as a discrete distribution with a few weights. In addition, these weights is represented with fewer bits.

3. Weights Sharing

Weights sharing is usually used with trained quantization to improve the quantization operation. Weights sharing operation generates a code book by clustering the weights into few clusters. This is done by storing few effective weights in a code book and let other weights share one of them. The operation flow can be summarized as shown in Figure 2.18. The operation starts be clustering the weights. This clustering is done by different methods such as k-means clustering. Secondly, a code book with the new clusters is generated. Then, quantization is made with the code book. Finally, retraining is required to recover the accuracy loss and then go back to quantization and so on.

Figure 2.19 shows a simple example for weights sharing operation. The example is made on a hidden layer neural network with 16 weights. Firstly, the weights is clustered into four groups with four different colors. Every cluster has its index which is used instead of using the value itself. By this way, only two bits are required to represent the index of the cluster. During the operation, the weight matrix on top left is converted to cluster index matrix. The weight matrix has a gradient matrix on bottom left which is grouped as shown on bottom middle matrix. Finally, the clusters centroids is updated using reduced gradient matrix and so on.

4. Huffman Coding

Huffman coding is a lossless data compression algorithm that can code the non-uniformly distributed values to save large memory storage. After quantizing the model, many values of the weights are repeated frequently. Algorithms such as Huffman coding [32] is utilized to represent the most frequent values in smaller bits and the least frequent values in larger bits. Combining pruning, quantization, and Huffman coding can obtain larger compression ratios.

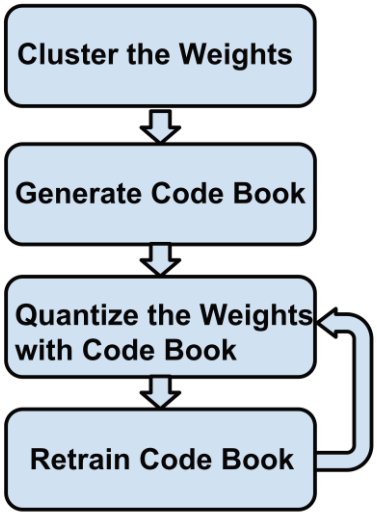


Figure 2.18: Weights sharing with quantization flow.

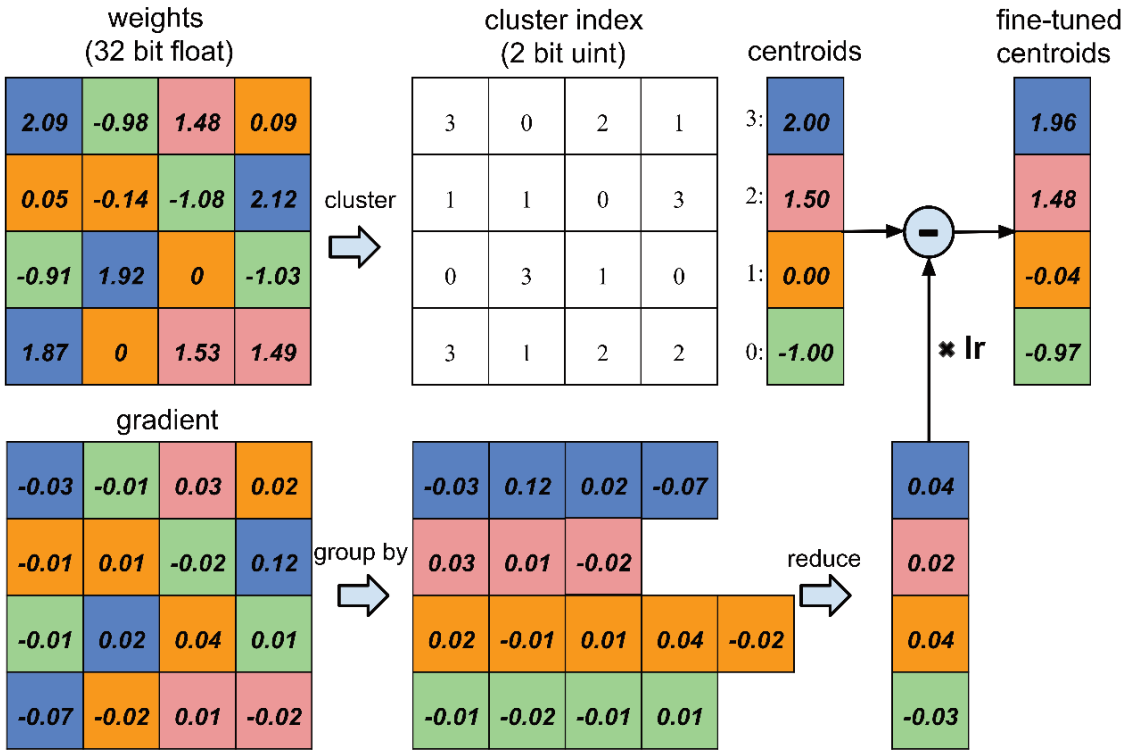


Figure 2.19: Weights sharing example.

5. Winograd Transformation

Winograd transformation is one of fast techniques to optimize the computation of convolutions [58]. It's simply transforms the convolution operation into point-wise multiplication by replicating the filter elements. This transformation saves the required multiplications for every single output. Figure 2.20 shows a simple example for Winograd transformation. The input feature map size is 4x4, while the filter size is 3x3, and the output feature map size is 2x2. If the conventional convolution operation is used, it requires 9 multiplications per single output element. Consequently, it required 36 multiplications per single output feature map with size 2x2. On the other hand, Winograd transformation extends the filter to 4x4 size and this saves the replicated multiplications during the convolution operation. Correspondingly, only 16 multiplication operations are required for 2x2 output feature map with 2.25x fewer multiplications.

6. Binary/Ternary Net

As the current deep neural networks consist of millions of parameters per model. Many researchers have worked on reducing the precision of every parameter to the least value as binary or ternary values. These neural networks are called Binary/Ternary net with two weight values for binary nets or three weight values for ternary nets. This algorithm is suitable for many applications that are trained on simple datasets like MNIST or CIFAR. However, there is another work that can apply it with ImageNet dataset and AlexNet CNN with acceptable top-1/top-5 error rates [59-60]. Figure 2.21 shows an example for neural network after retraining for ternary weights.

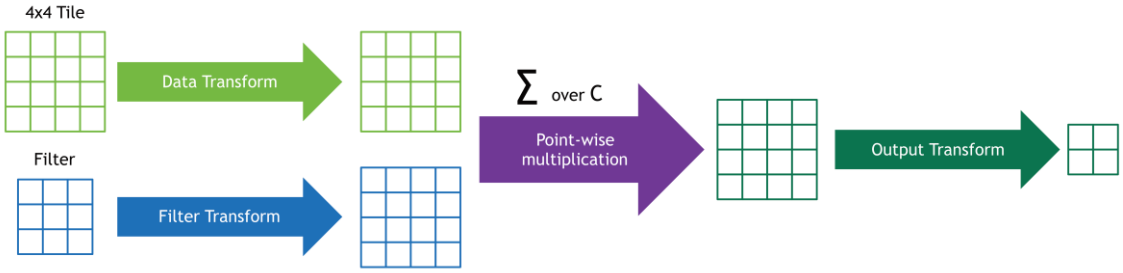


Figure 2.20: Example for Winograd transformation.

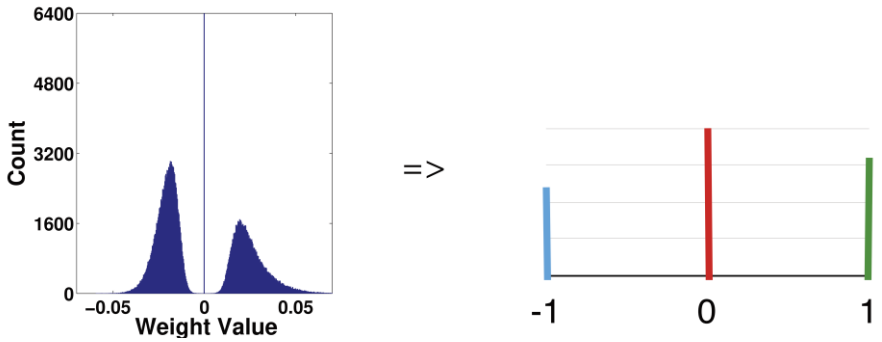


Figure 2.21: Retraining neural networks for ternary weights.

7. Low Rank Approximation

During the past decade, the size and depth of convolution neural networks tend to increase to get higher classification accuracies and solve more complex problems. However, the computational cost of these CNNs also increases significantly. The computational cost makes the training and testing phases more complex. Low rank approximation technique can be used to reduce this complexity. This is made by decomposing large convolutional layer with d filters with filter size of k × k × c, where k is the spatial size of the filter and c is the number of input channels of this layer. It's decomposed from one layer as shown in Figure 2.22 (a) to two layers as shown in Figure 2.22 (b)

- A layer with d' filters (k × k × c)
- A layer with d filter (1 × 1 × d')

By applying this technique, the computation complexity can be reduced from $O(dk^2c)\ to\ O(d'k^2c) + O(dd')$. Correspondingly, the computations can be speed up multiple times. In addition, this technique is proved to increase the classification accuracy as well [61].

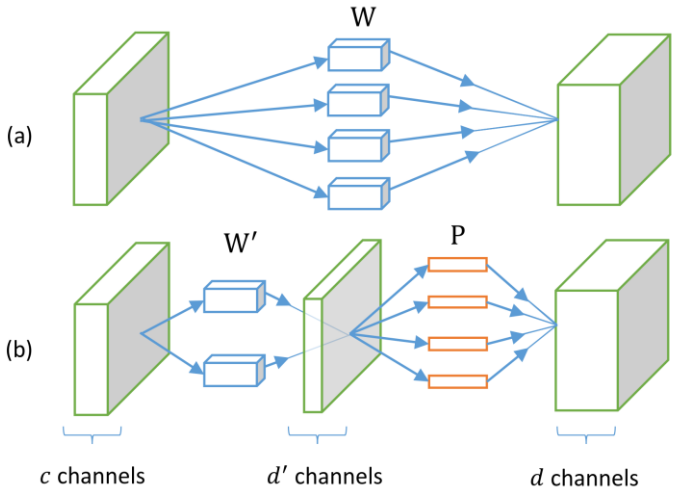


Figure 2.22: Low rank approximation example. (a) Original layer, (b) approximated layer [61].

2.3. Hardware Design

In this section, common hardware design options are studied by reviewing several implementations. Secondly, Popular AI ASIC chips are presented to give an example for this type of implemenation. Moreover, FPGA overview is presented to show the main building blocks and design flows. Finally, an overview about fixed-point representation is presented as it will be used in the proposed hardware accelerator.

Hardware implementation options are categorized into different categories. The first category is the general purpose hardware, which includes both CPUs and GPUs.

Secondly, specialized hardware category which has different types of categories, but it's mainly divided into ASIC chips and FPGA implementations. Every category has its features, advantages, disadvantages, limitations, and others. Table 2.2 shows a summary and comparison between all of them.

If CPUs, GPUs, FPGAs, and ASICs are compared in terms of flexibility, it is found that the CPUs and GPUs are better due to the availability for supporting programming languages and frameworks as shown in Figure 2.23. In contrast, ASICs are less flexible as they require custom frameworks. While FPGAs are currently supported by several framework such as OpenCL, but they require more development time. On the other hand, the efficiency is the best while using ASICs as it uses fixed/custom logic as shown in Figure 2.23. FPGAs come at next efficient type, but CPUs are the least flexible hardware as it is designed as a general purpose platform.

Table 2.2: Comparison between AI hardware design methods

Design Type	**CPU**	**GPU**	**FPGA**	**ASIC**
Main Features	Traditional sequential processor	Parallel cores for graphics processing	Configurable logic gates and IP cores	Optimized integrated circuits for specific application
Power Consumption	High	High	Medium	Low
Strengths	Handling complex instructions	Highly parallel cores used not only for graphics processing but also for AI processing, a lot of supporting frameworks for AI	Flexibility, reconfigurability, specific design, variety of resources: LUTs, DSPs, etc.	Low power consumption, speed, low footprint
Constraints	Memory access bottlenecks, few parallel cores	High power consumption, large foot print	Programming complexity	High Cost, fixed logic, large time-to-market
Programming	Assembly languages, high level languages	OpenCL, C, C++, Python, Nvidia CUDA	Verilog, VHDL, OpenCL, HLS	Custom programming

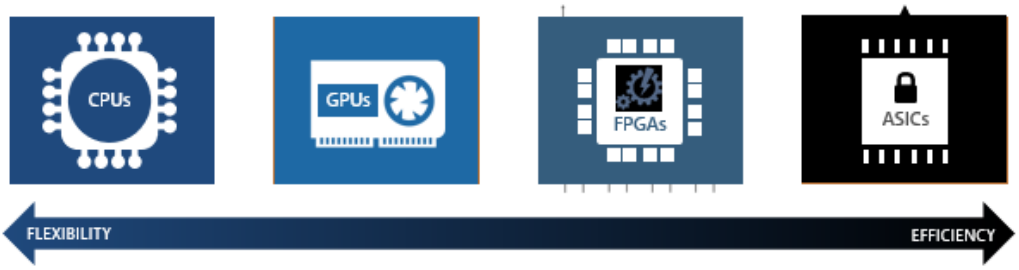


Figure 2.23: Flexibility versus efficiency for CPUs, GPUs, FPGAs, and ASICs

2.3.1. FPGAs Overview

FPGAs are one of the common hardware design methods that provide high parallelism, optimized hardware, and real-time computation capabilities. FPGAs consist of programmable logic gates and interconnections that enables reconfiguring them to do specific functions. HDL (hardware descriptive languages) such as Verilog or VHDL are mainly used to make a design on FPGA. However, there are different design flows on FPGA like HLS (High Level Synthesis) and OpenCL (Open Computing Languages). In this section, a quick overview about FPGA main blocks are presented, and FPGA design flow is illustrated.

FPGAs contain an array of configurable logic blocks (CLBs), and a hierarchy of reconfigurable interconnects that allow the blocks to be routed together. Also, FPGAs contain memory elements, which may be simple flip-flops or more complete blocks such as block RAMs or ultra-block RAMs. Figure 2.24 shows a simple schematic for FPGA internal design.

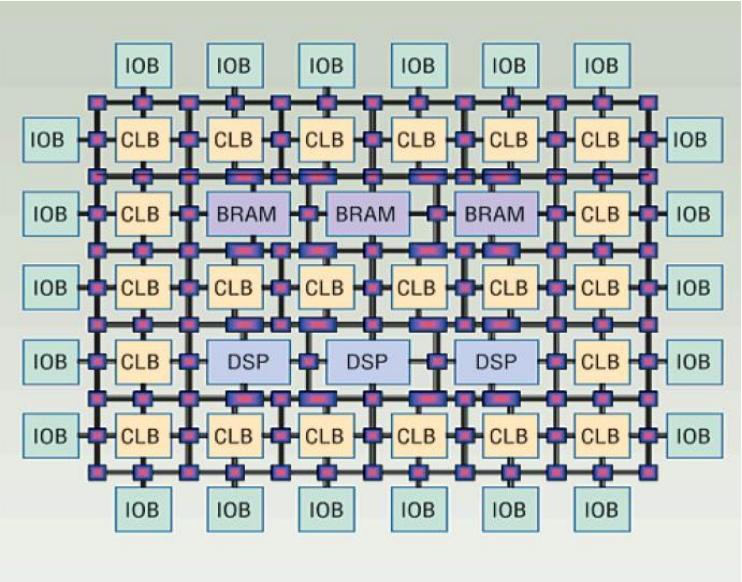


Figure 2.24: FPGA Internal Design

- Configurable Logic Blocks (CLBs)

CLBs carries the logic for the FPGA. The block contains lookup tables (LUTs) for creating arbitrary combinatorial logic functions, which are made of ROMs. Also, it contains flip-flops for clocked storage elements, along with multiplexers in order to route the logic within the block and from/to external resources. Figure 2.25 shows an example for basic CLB block. For the modern FPGA today, CLBs contain enough logic to create a small state machine.

- Block RAM

It is a dedicated block RAM for memory storage on FPGA without using FPGA LUTs. It serves as a relatively large memory structure, but much smaller than off chip memory resources.

- DSP Cores

Digital Signal Processors (DSPs) are used in FPGAs for complex arithmetic functions. They are specialized processors that are used to implement Multiply Accumulate blocks in addition to video and audio processing.

- Programmable Interconnections

They are the main routes that can be used to connect CLBs with each other on the FPGA. These interconnections are used as buses within the chip as shown in Figure 2.26. Transistors are used to turn on/off connections between different blocks. Programmable switch matrices in the FPGA are used to connect the long and short interconnections together. In addition, there are global clock interconnects which are specially designed for low impedance and fast propagation clocks. These interconnects are used to connect the clock buffers to clocked element in each CLB.

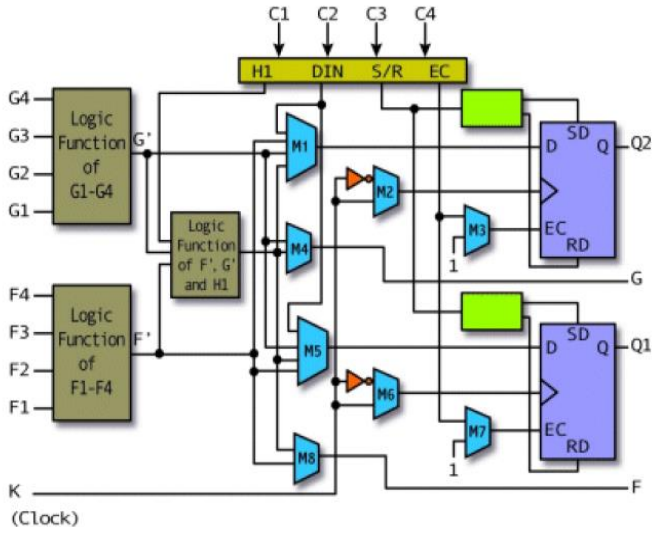


Figure 2.25: Example of Configurable Logic Block (CLB)

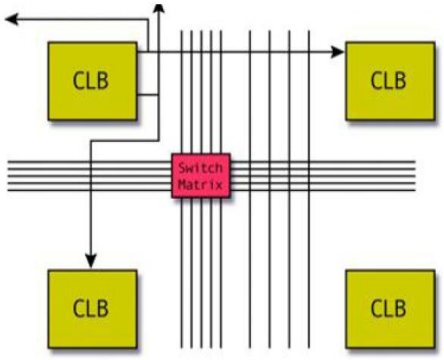


Figure 2.26: FPGA programmable interconnections

- Configurable I/O Blocks

A Configurable input/output (I/O) Blocks are used for input/output off-chip connections. It consists of an input buffer and an output buffer with three-state and open collector output controls as shown in Figure 2.27. Typically, there are pull up resistors on the outputs and sometimes pull down resistors that can be used to terminate signals and buses without requiring discrete resistors external to the chip. The polarity of the output can usually be programmed for active high or active low output.

- Clock Circuitry

Clock circuitry is a special I/O block with special high drive clock buffers, known as clock drivers. These buffers are distributed around the chip to connect drive the clock signals onto the global clock lines. These clock lines are designed for low skew and fast propagation.

- Embedded Cores

Embedded cores are added by the FPGA vendor as separate blocks to provide more peripherals for the developers. The performance of these core do not depend on the rest of the design since it doesn't need to be placed and routed.

- **FPGA Design Flow**

Figure 2.28 shows FPGA design flow. The flow is summarized as follows:

1. Functional Specifications: system-level design is set and all specifications are determined.
2. HDL coding: the HDL code is written, and then behavioral simulation is done to make sure of the design functionality.
3. Synthesis: HDL is elaborated and synthesized into logic gates. This is intermediate state before doing place & route step. At this step, static timing analysis and estimate power consumption in addition to design utilization can be calculated.
4. Place & Route: The design blocks and logic cells are placed on the FPGA are routed together.
5. Download the bit stream: The HDL code is burned on the FPGA.

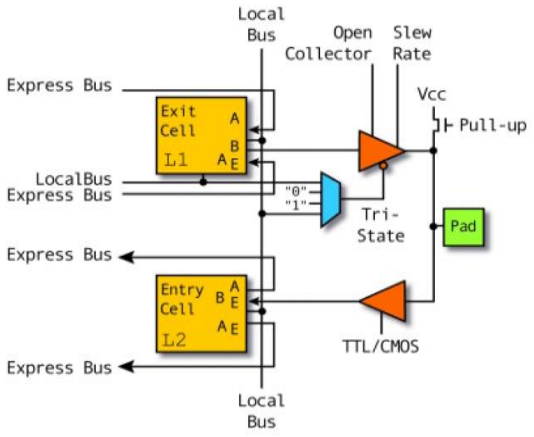


Figure 2.27: FPGA configurable I/Os.

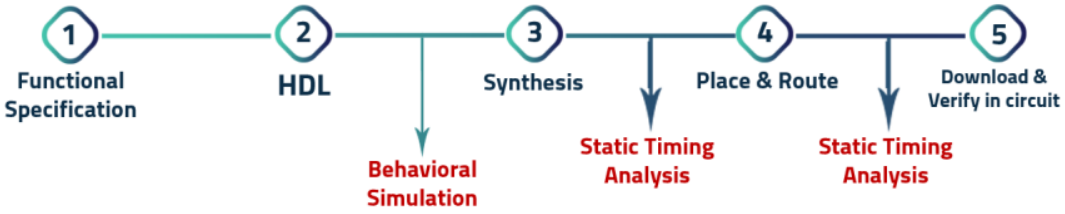


Figure 2.28: FPGA Programmable interconnect

2.4. Literature Review

Hardware accelerators get a huge attention in different research areas through past years. Researches compete to propose the best architecture or implementation for different Deep Learning applications. As the proposed work is focused on CNNs, the popular CNN hardware accelerators are investigated. The review firstly investigates the popular FPGA implementations. Secondly, ASIC hardware accelerators are presented. Finally, A review for previous GoogLeNet CNN hardware accelerators. After presenting the review it becomes helpful to set the proposed architecture while avoiding drawbacks of previous architecture and adding new features.

- FPGA Implementations

There are two popular high-level design flows of hardware accelerator implementation. The first flow is high-level synthesis (HLS), and the second one is Open Computing Language (OpenCL). They provide fast and easy hardware implementation, but they have a lack of optimization and energy efficiency. These high-level flows have been developed to build programs and execute them across heterogeneous platforms, such as CPUs, GPUs, and FPGAs [22-24]. HLS is an automated process to compile digital hardware circuits by synthesizing them. It enables building and verifying the hardware by giving better control over the architecture [25-27].

Aydonat proposes a new architecture written in OpenCL that minimizes external memory bandwidth and maximizes data reuse [22]. Furthermore, Winograd algorithm is used to increase the data reuse and decrease the number of computations. The design is implemented on Intel's Arria 10, and processes 1382 GFLOPs. Aydonat's hardware accelerator achieves a performance of 23fps/W when running the AlexNet.

Jialiang's hardware accelerator is another CNN hardware implementation based on OpenCL on FPGA [23]. The design is implemented on Altera Arria 10 GX1150. It achieves 866GOP/s floating-point performance at a frequency of 370MHz and 1.79TOP/s fixed-point performance at a frequency of 385MHz.VGG is processed as a case study with 28.1fps in floating-point representation.

Suda's implementation is another OpenCL-based design on FPGA. It achieves a peak performance of 136.5 GOPS for convolution operation, and 117.8 GOPS for the entire VGG network that performs ImageNet classification. Both AlexNet and VGG are tested on this design using two Altera Stratix-V FPGA platforms, DE5-Net and P395-D8 boards, which have different hardware resources.

Zhang proposes a hardware accelerator for deep convolution neural networks [25]. This hardware accelerator is HLS design implemented on the VC707 FPGA board. In addition, Software implementation runs on an Intel Xeon CPU E5-2430 with 15MB cache. It achieves a peak performance of 61.62 GFLOPS at a frequency of 100MHz with 18.6W FPGA power consumption.

fgpaConvNet is a framework for CNNs on FPGA based on HLS design [26]. This fraemwork introduces FPGA reconfiguration as a design option for CNNs FPGA implementations. The design is implemented on Zynq-7000 XC7Z020 FPGA at a frequency of 100MHz. fgpaConvNet achieves 12.73GOP/s and 7.27 GOP/s/W, and supports fixed-point as well as single and double precision floating-point representations.

FINN is a framework for building fast and flexible hardware accelerators using a flexible heterogeneous architecture [27]. It is designed especially for binary neural networks on ZC706 embedded FPGA platform while consuming less than 25W total system power. FINN has a 0.31µs latency on the MNIST dataset with 95.8% accuracy, and 283µs latency on the CIFAR-10 and SVHN datasets with 80.1% and 94.9% accuracy, respectively.

Moreover, many previous studies focus on accelerating the convolution layers of CNN only. For example, in [28] and [29], the hardware accelerator processes several convolution layers only rather than the full CNN while neglecting other CNN layers, such as fully connected layers. Consequently, those accelerators are not suitable to be deployed in low-power embedded applications.

- ASIC Implementations

EIE is an energy efficient inference engine that is developed by Stanford [64]. EIE provides 120× energy saving more than the conventional processors. EIE has a peak performance of 102GOP/s. It's designed to work directly on a compressed CNN, but it can process an uncompressed network with 3TOP/s. power consumption is 600mW.

EIE has two versions with 64PEs and 256PEs. The first version operates at a frequency of 800MHz with power consumption 0.59W. The chip area is 40.8mm^2 with 45nm technology and fixed-point 4-bit. The second version operates at a frequency of 1200MHz with power consumption 2.36W. The chip area is 63.8mm^2 with 28nm technology and fixed-point 4-bit.

Eyeriss v1 is developed by MIT [63]. It uses row stationary (RS) datafow with 168 processing elements. Eyeriss processes the convolutional layers at 35fps for AlexNet at 278mW with batch size 4, and 0.7fps for VGG-16 at 236 mW with batch size 3. Chip size is 16mm^2 with TSMC 65nm technology. The chip core operates from 100MHz to 250MHz with a peak throughput 16.8 to 42 GOP/s

Eyeriss v2 is the second generation for Eyeriss acelerator [21]. RS dataflow is upgraded to RS+ dataflow with many improvements. A network-on-chip (NoC) architecture is used for both multicast and point-to-point single-cycle data delivery. The number of PEs can be varied and increased from 256 PEs to 16384 PEs. Eyeriss v2 shows a performance increase between 10:17× for 256 PEs, 37:71× for 1024 PEs, and 448:1086× for 16384 PEs.

DianNao accelerator is designed using CMOS technology of 65nm with an area of 3.02mm2 [20]. It performs 452GOP/s of fixed-point operations in parallel with 0.485W (excluding main memory accesses). This accelerator is 21.1x more energy-efficient than a 128-bit SIMD core, and operates at 2GHz. However, the reported throughput is the peak theoretical throughput only for some convolution layers without DRAM access time, which degrades the speed and increases the power consumption.

- GoogLeNet Hardware Implementations

Snowflake accelerator [18] is able to achieve an average computational efficiency of 91%, and is implemented on a Xilinx Zynq XC7Z045 APSoC. Snowflake is capable of achieving 128GOP/s while consuming 9.48W of power. This work considers the number of frames without the fully connected layers. Correspondingly, adding the fully connected layers overhead degrades its throughput and increases its power consumption. Moreover, it has high power consumption due to the usage of 1GB of DDR3 memory in addition to two ARM cores running at 800MHz and one Kintex-7 FPGA. The entire design is operated at a frequency of 250MHz.

Another hardware accelerator that is designed by Zhao is synthesized by using the TSMC 65nm CMOS technology and achieves a peak of 280.8GOPS/s [19]. Its core area is 4.35mm^2 running at 650MHz with a power dissipation of 859mW. Convolution layers implementation of popular CNNs shows a frame rate of 36.7fps for ResNet-34 and 179.5fps for AlexNet. Compared with the existing AlexNet accelerators reported in recent years, this accelerator achieves 3.1x average area efficiency, 1.7x energy efficiency, and 20% higher average computational efficiency. However, the input image/feature data and filter weight parameters are transferred from the external off-chip memory to the separated on-chip data buffer and parameter buffer. In addition, this work considers the number of frames without the fully connected layers similar to [18], and correspondingly adding this overhead degrades the throughput and increases its power consumption.

CoNNA is another hardware accelerator that processes different types of CNNs specially GoogLeNet [53]. CoNNA is implemented on Xilinx ZCU102 with three different versions using different resources and operating frequencies. In contrast to most existing solutions, CoNNA process fully compressed CNN models, which gives it more advantages than using uncompressed CNN models. CoNNA is designed as a reconfigurable architecture that operates at 60MHz, 100MHz, and 200MHz frequencies. CoNNA_C3 is one of CoNNA versions that reaches a peak performance with 17.325GOP/s and classifies 4.95fps for GoogLeNet.

The last implementation is Kalle inception module [62]. As discussed before, GoogLeNet consists of 9 inception modules in addition to multiple layers such as 7x7 convolution, LRN, Averagepooling, fully connected, Softmax, and maxpooling layers. But this work carries the implementation for inception module with size 14x14 on Xilinx Artix7A200 FPGA. It achieves peak performance with 9.92 GFLOPS at 100MHz.

Chapter 3 : Memory Compression

3.1. Introduction

Increasing the model size has become a common trend within the development of CNN models. These models have a huge number of weights that require large memory storage. As stated by [30], 32-bit DRAM memory access requires 640pJ, which leads to a fast battery drain of the embedded devices. Model compression techniques such as weights pruning and weights quantization are improved to be deployed in these CNNs models. On the other hand, deep neural networks consist of a dramatically large number of connections between the neurons, which makes the model contains millions of parameters. Many of these connections are not important, and removing them yields a large compression of the model. Weights pruning is a processing operation that removes unnecessary connections. Removing these connections degrades the model accuracy as the model connections are mutually dependent. Correspondingly, retraining the remaining connections is a mandatory step to recover the accuracy loss as in [30]. Consequently, model compression is applied for any type of deep learning models with little accuracy degradation.

3.2. Related Work

Weights of deep learning models are usually represented with 32-bit precision, which allocates a large size of memory. Representing these weights in a smaller precision can compress the model significantly. Gong and Yunchao propose a method for quantizing the weights during training using a codebook that results in a smaller representation for the weights in terms of precision [31]. Moreover, quantizing the weights reduces the accuracy directly, and accordingly, retraining is necessary for quantization as the same as weights pruning.

After quantizing the model, many values of the weights are repeated frequently. Algorithms such as Huffman coding [32] is utilized to represent the most frequent values in smaller bits and the least frequent values in larger bits. Combining pruning, quantization, and Huffman coding can obtain larger compression ratios. Applying memory compression on neural networks is an open area of research. Many techniques are proposed to deal with different models and achieve higher compression ratios.

In [30], a pruning pipeline is proposed that firstly retrains the model from scratch, then performs the weights pruning iteratively, and retrains to compensate for the accuracy loss due to the reduction of weights count. However, this model takes a large retraining time due to its iterative pruning and training. Moreover, there is no chance for the removed connections to be restored when it is found that they are essential. On the other hand, [33] proposes a method of pruning and splicing the connections simultaneously. By splicing, the removed connections can be restored. Moreover, its running time is much shorter than previous work. Moreover, there are other pruning methods that dynamically prune the channels of the network based on the input as in [34]. For network compression, [35] proposes a method of weights incremental quantization. The method operates till all weights become either zeros or power of 2's. As the pruning is applied before quantization, increasing the levels near zero can

further improve the compression results. Finally, compression pipelines are proposed in [36], which consists of pruning, quantization, and Huffman coding. This helps to achieve a large compression ratio.

3.3. GoogLeNet CNN

The proposed processor is designed to fit GoogLeNet inception CNN [17]. GoogLeNet CNN achieves higher inference accuracy while keeping the weights count of ~6.9 Million only, which is a great improvement compared to previous CNNs. Moreover, the size of weights is cut down significantly. For many years, it is well-known that the depth of the network should be increased to get higher accuracies, especially the number of convolution layers. This has been a common direction till year 2014 when Szegedy proposed a new CNN network called GoogLeNet with the concept of inception module as shown in Figure 3.1. In GoogLeNet, the depth and width of the network have been increased, but the computational budget has been kept constant by using the network-in-network concept. This concept uses additional 1x1 convolutional layers to remove the network bottlenecks to help in dimension reduction. GoogLeNet overcomes legacy CNNs such as AlexNet and VGG by getting the highest accuracy with fewer weights.

GoogLeNet has 57 convolution layers and only one fully connected layer. The computation workload is centered in convolution layers with 2.58G MACs. Furthermore, the fully connected layer uses a huge number of weights per layer with 1.024M weights. Moreover, it has fourteen Maxpooling layers to reduce the input feature map size. The network has one average pooling layer to reduce the input feature map size before the fully connected layer. Finally, the softmax layer is used to get the classification results in probabilistic values. Table 3.1 lists the detailed architecture and design parameters of GoogLeNet.

3.4. GoogLeNet Training

GoogLeNet is built based on Szegedy work [17]. The network structure is built as shown in Figure 3.1 with all layer sizes as mentioned in Appendix A. It is trained for 100 epochs on ImageNet Dataset. ImageNet is one of the most popular datasets which have more than 1000 classes with 14 Million training images as mentioned in Appendix B. Furthermore, the optimization is done with stochastic gradient descent using a learning rate of 0.01, a momentum of 0.9, and a weight decay of 10^{-4}. Every 30 epochs, the learning rate is divided by 10. The training accuracy is presented in the results section.

3.5. Compression Model

GoogLeNet model is compressed with a combined framework of weights pruning and quantization. The proposed framework consists of two stages which are selected carefully after exploring all related memory compression methods. Firstly, the framework applies the weights pruning based on dynamic network surgery work [33]. Secondly, the proposed hyper-framework quantizes the network iteratively based on the

Table 3.1: GoogLeNet analysis

GoogLeNet CNN	Count
Convolution layers	57
Convolution layers in depth	21
Convolution workload (MACs)	2.58G
Convolution parameters	5.9M
Activation layer	ReLU
Maxpooling layers	14
Average pooling layers	1
FC layers	1
FC workload (MACs)	1.024M
FC parameters	1.024M
Total workload (MACs)	2.58G
Total parameters	~6.9M

incremental network quantization (INQ) framework [35]. The proposed framework is built without applying Huffman coding to avoid overhead latency of Huffman decoding while fetching the weights on the FPGA hardware. Figure 3.2 shows a summarized flow chart for the used hyper-framework. Every framework for both weights pruning and weights quantization will be discussed clearly in the following two sections:

3.5.1. Weights Pruning

Weight pruning is performed using a dynamic network surgery method [33]. Unlike the previous methods of alternating pruning and retraining, the dynamic network surgery method performs connections pruning and splicing for the network iteratively and implements the whole process dynamically. The method is tested before on smaller datasets like MNIST and other CNN Models such as LeNet and AlexNet, but it is applied for the first time on ImageNet dataset and GoogLeNet CNN model.

Weights pruning is performed on both, convolution layers and fully connected layers into two steps. Firstly, convolution layers are pruned successfully, secondly the fully connected layer. Dividing weights pruning operation is important to keep the accuracy as it is proved experimentally that performing weights pruning in one step causes some degradation on the overall accuracy. Dynamic network surgery method is performed by applying pruning and splicing for the network iteratively as shown in Figure 3.2. Firstly, activation masks are initialized for all weights to activate all of them. The masks are set during the process to one or zero to activate or deactivate them, respectively. During the forward-propagation, the masks are element-wise multiplied by the weights, and the resulting outputs are used in the network. During splicing, the values of the masks change according to weights mean, and standard deviation. As a result, they might be reactivated for some weights to recover the connections that are found to be important during retraining. This results in making accuracy degradation insignificant. By using this method, a lot of model parameters are trimmed and the classification accuracy will not be hurt too much.

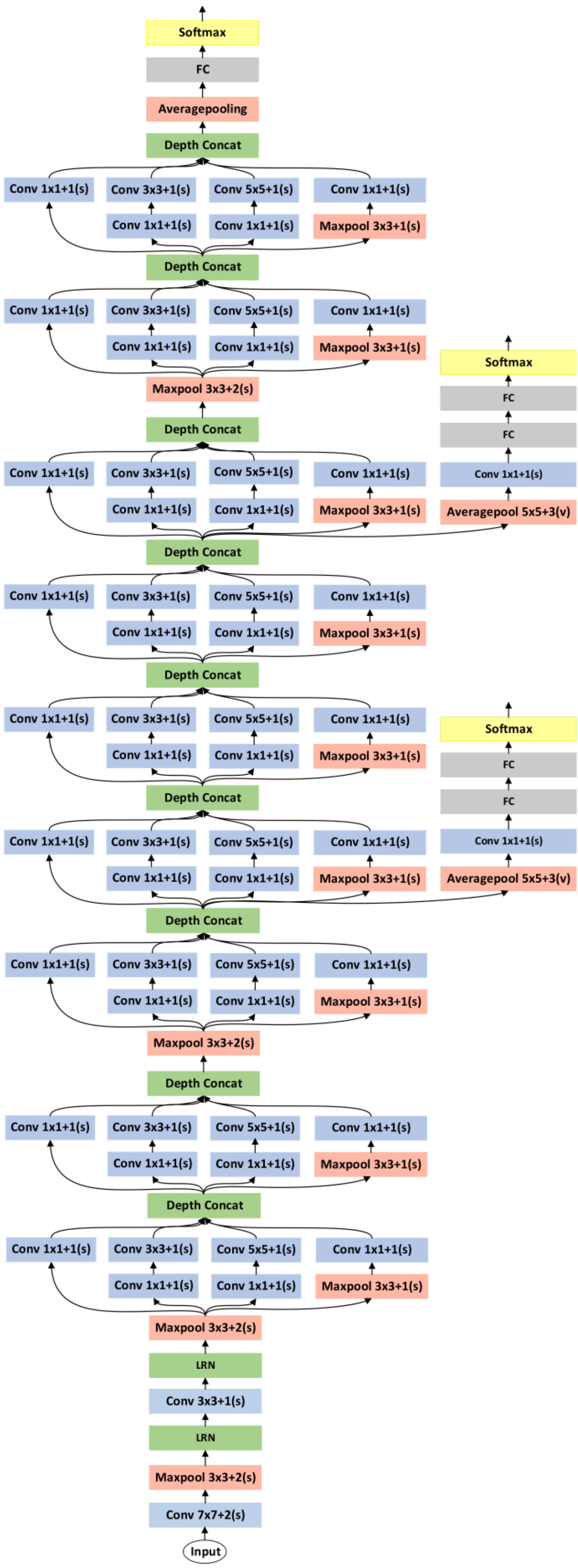


Figure 3.1: GoogLeNet CNN network structure [17]

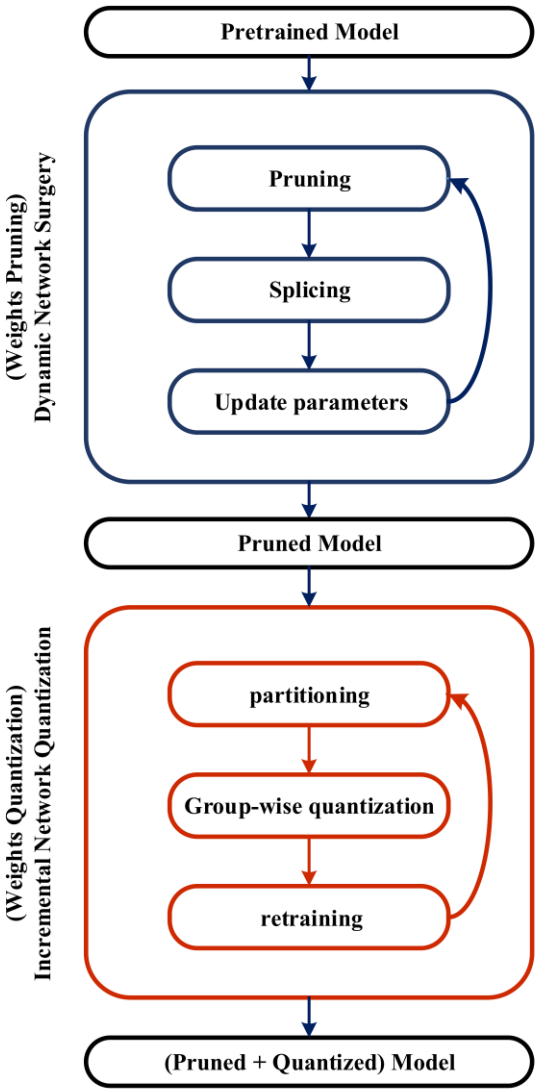


Figure 3.2: The proposed memory compression model

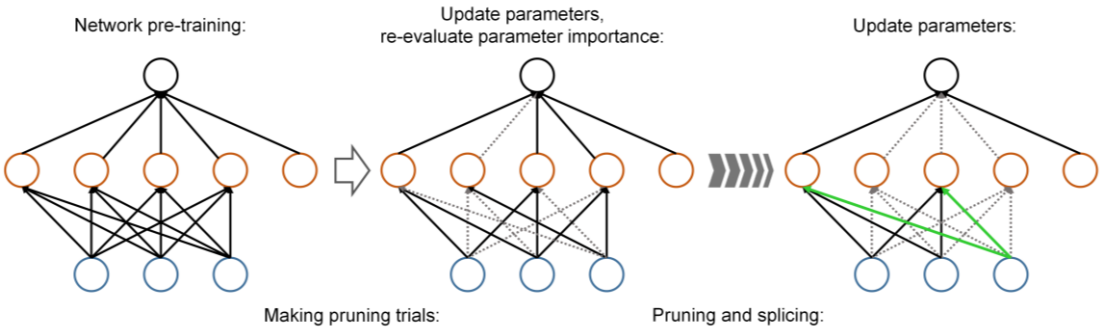


Figure 3.3: Dynamic network surgery pruning method steps [33]

3.5.2. Weights Quantization

After applying weights pruning model, weights quantization is used to shrink the precision from 32-bit to 4-bit. After analyzing multiple quantization frameworks, Incremental Network Quantization (INQ) framework is used [35]. INQ is a group-wise quantization that is performed by partitioning the weights into two groups iteratively. Weights partitioning uses a pruning-inspired measure to split the two groups in each layer based on their values. The first group is quantized to the target precision, and the second group is retrained to compensate for the accuracy loss. Weights are iteratively quantized to 4-bit with a value of zero or a number with a power of 2's. All the weights that tend to zero are quantized to zeros to keep the effect of the network pruning. The number of quantization steps is increased at the end to avoid the sudden accuracy loss at the end of the quantization. The process is simplified in Figure 3.4 where the pretrained connections are colored with black, quantized connections are colored with green, and retrained connections are colored with blue. Also, operation (1) represents a single run of group-wise quantization and retraining. Moreover, operation (2) denotes the repeating operation of operation (1). INQ iterates with the assigned steps where every step takes some percentage of the weights to quantize it.

3.6. Compression Results

In this section, compression results are presented and some experiments on compressing GoogLeNet are demonstrated. The model is firstly trained that trained with ImageNet dataset based on [17] work, yielding a top-1 accuracy of 71.39%. The reference model has ~6.9M weights with 32-bit precision. The training is made as discussed in section 3.3. Secondly, weights pruning is made with dynamic network surgery model as discussed in section 3.4.1. The model is pruned to have less than 1 million parameters only to fit in Virtex-7 FPGA without using off-chip DRAMs. Consequently, the pruning is made aggressively to reach 7.2x compression ratio with a top-5 error rate of 1.4% and top-1 error rate of 2.7% as listed in Table 3.2.

Finally, incremental network quantization is applied to the pruned model to reduce the precision of weights from 32-bit to 4-bit. The removed connections are suppressed to zeros, and the quantization is performed iteratively on the remaining weights. At first, the accumulated partitions of quantized weights at iterative steps are set as reference paper as [0.2, 0.4, 0.6, 0.8, 1], but there is a sudden drop in the classification accuracy with 10% in top-1 accuracy.

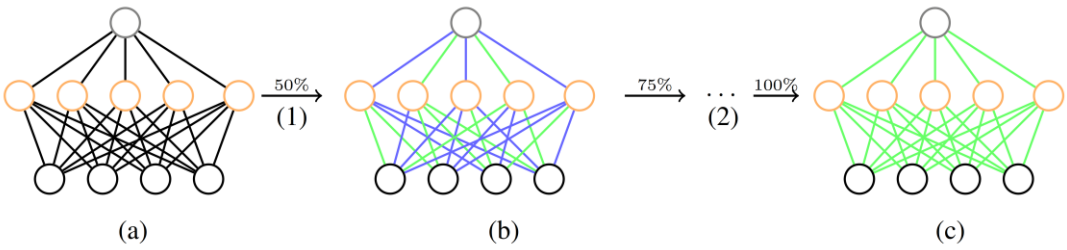


Figure 3.4: Incremental network quantization steps [35]. (a) pretrained/full-precision model, (b) updated model after one iteration, (c) Final quantized model

Table 3.2: Error rates and compression ratio for different compression models

Model	Top-1 error rate (%)	Top-3 error rate (%)	Top-5 error rate (%)	Compression ratio
Reference model	0	0	0	1x
Pruned model	2.7	1.7	1.4	7.2x
Hyper model	4.8	3.5	2.6	57.6x

this sudden drop happened because the model has many sparse weights and GoogLeNet has large network width with fewer parameters than other CNNs such as AlexNet and VGG. Therefore, the last steps starting from 80% quantization are increased to quantize and retrain the remaining weights carefully. Consequently, the model is quantized using percentages of [0.2, 0.4, 0.6, 0.8, 0.85, 0.9, 0.95, 1], which yields a loss of 4.8% for top-1 error rate and 2.6% for top-5 error rate as listed in Table 3.2. Quantizing from 32-bit to 4-bit leads to a compression ratio of 8x independently. Correspondingly, the hyper model of the weights pruning followed by quantization compresses the model with 57.6x successfully, as shown in the compression chart in Figure 3.5.

Figure 3.6 shows the weights size reduction for each layer in the GoogLeNet model for the plain model, pruned model, and quantized model with colors blue, gray, and orange respectively. It is observed from the chart how the pruning firstly reduces the number of weights with gray columns. Secondly, quantization makes a reduction with 8x for every layer which is shown with the orange columns. Moreover, the fully connected layer is the most compressed layer, as it has a huge number of weights that tend to zero. On the other hand, 3x3 convolution layers come at the second most compressed layers due to the large number of filters they have. Memory reduction for every layer in Figure 3.1 is observed from Figure 3.6 by linking its name and location from the CNN network to the weights compression chart.

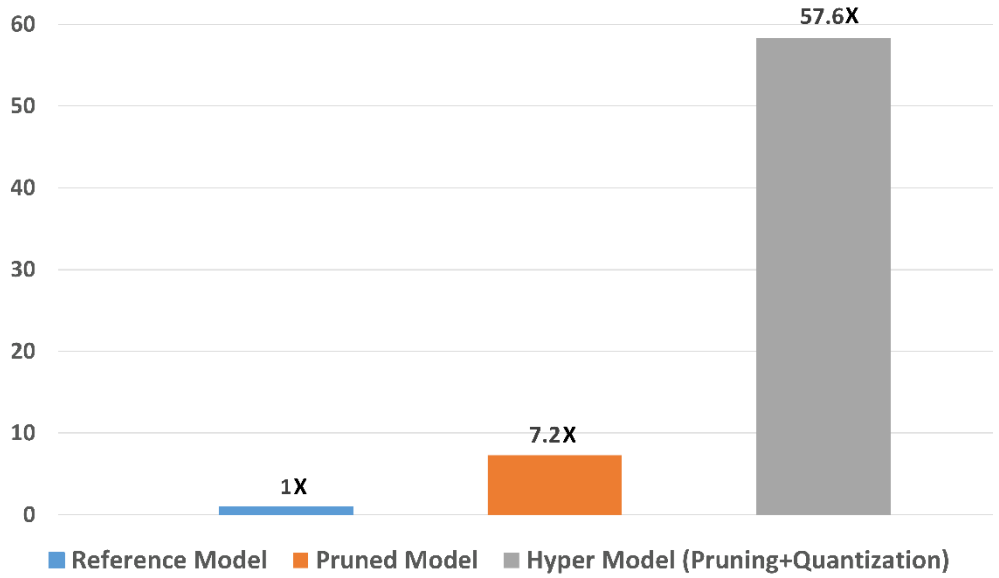


Figure 3.5: Compression ratios after pruning and quantization

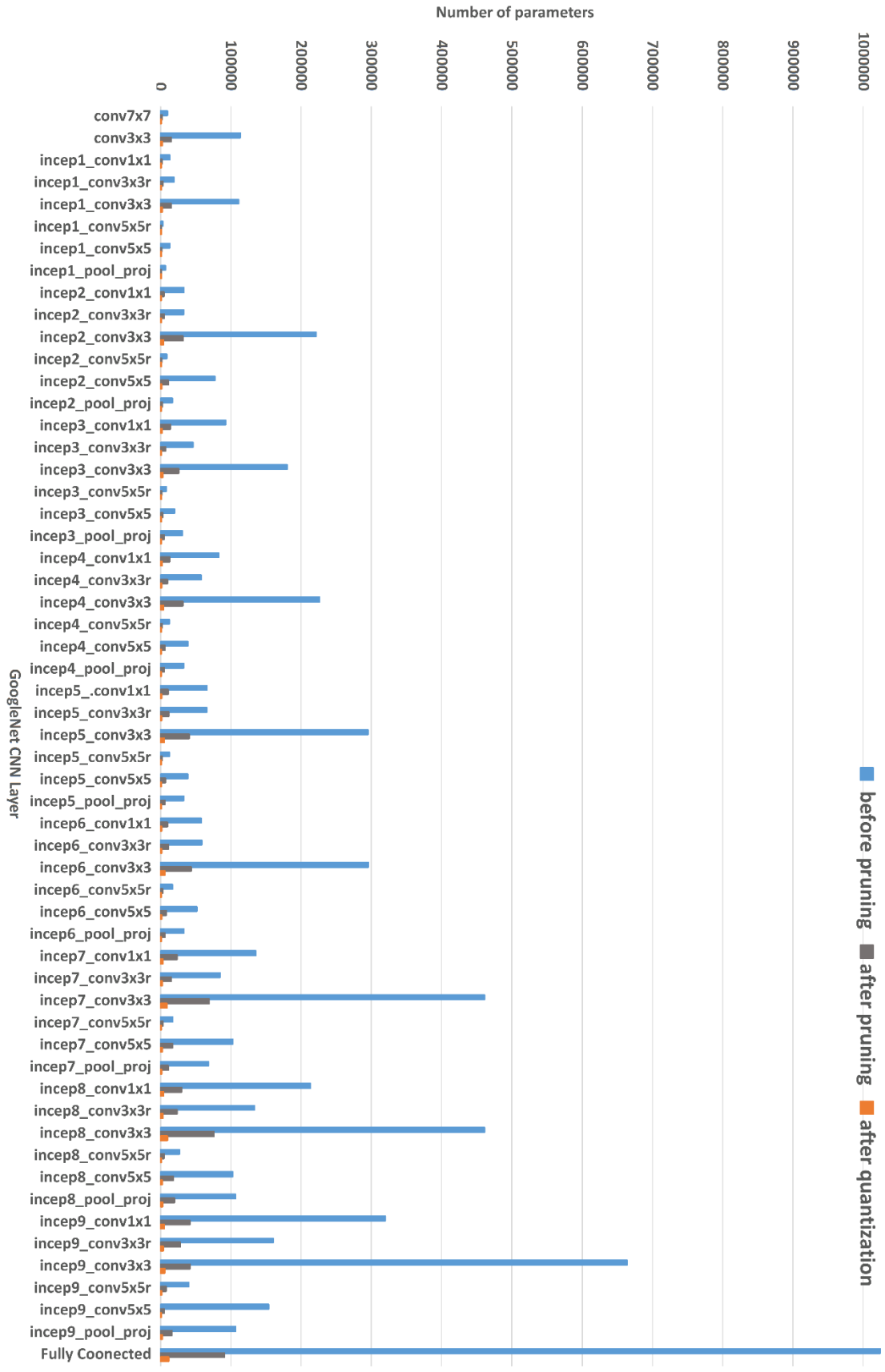


Figure 3.6: Weights compression for GoogLeNet layers

Chapter 4 : Architectural Design and Implementation

This chapter presents the design and implementation for the proposed hardware accelerator. The chapter gradually builds up the complete image of the full architecture. This is done by presenting each block by showing its main features. The architecture is built as a time-sharing processor that performs the computations for layers, batch by batch. The processing flow is made depending on the accelerator's control units and CNN structure. The adopted parallelism techniques and loop tiling are firstly presented. Secondly, the design of each unit is discussed in this chapter by showing its specs and implementation. Some of the units are modified with new improvements. These modifications are presented by comparing it with the older version. In addition, the important data flow is presented to show how the data is moved and handled between each unit. At the end of the chapter, several general modifications are made to improve the proposed accelerator and make full use of observed enhancements after memory compression results.

The proposed state-of-art processor consists of 256 memory banks, 224 parallel elements for multiplications, weights memory, accumulator unit, Maxpooling unit, average pooling unit, fully connected unit, softmax unit, buffers, and nine distributed control units. Each unit is carefully designed and implemented in native RTL (Register Transfer Logic) (Verilog) to achieve the best performance taking into account the power consumption. The top-level diagram of the proposed architecture is shown in Figure 4.1.

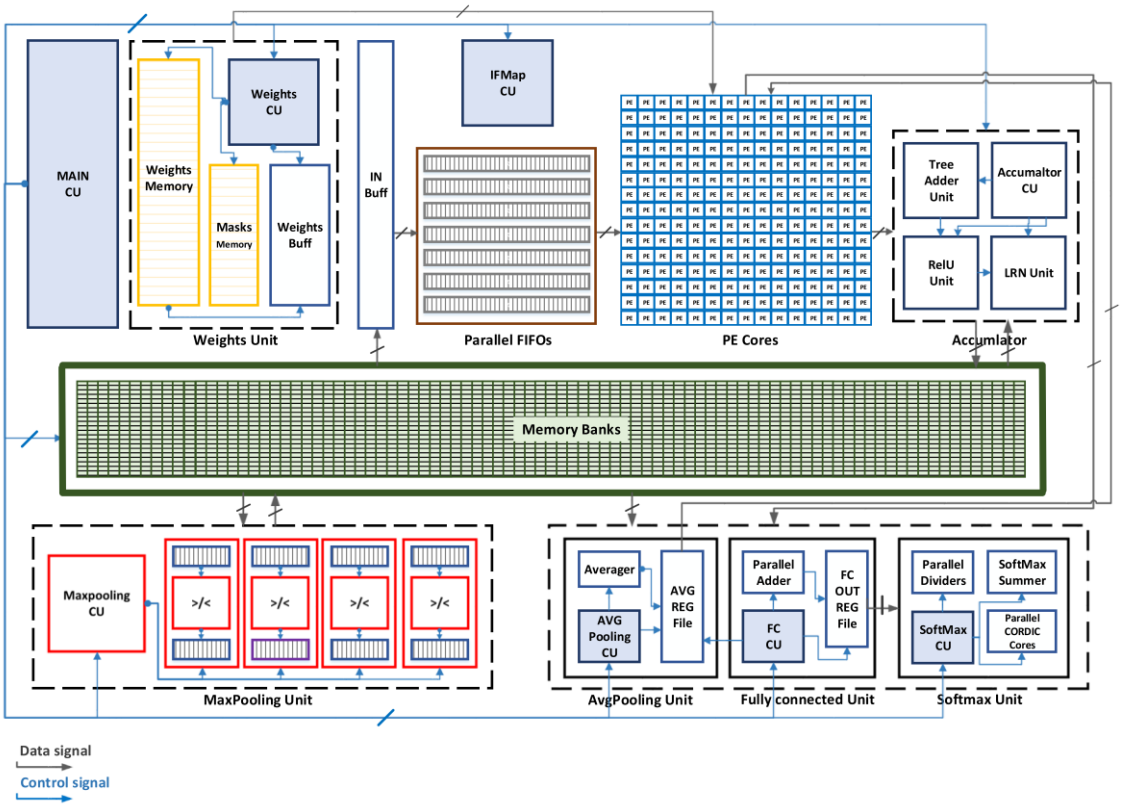


Figure 4.1: Top-level diagram of the proposed architecture

4.1. Parallelism

As the CNNs have a huge number of computations in convolution and fully connected layers. Computations parallelism is required to reach a short inference and training times. There are different ways of parallelism in CNNs, such as Batch parallelism, inter-layer parallelism, inter-feature map parallelism, inter-convolution parallelism, and intra-convolution parallelism, as stated by [14]. Every hardware accelerator adopts one or more of these types to speed up the processing. In the proposed accelerator, 24 kernels of 3x3 convolution layers, nine kernels of 5x5 convolution layers, or four kernels of 7x7 convolution layers, are processed in parallel as shown in Figure 4.2 and listed in Table 4.1. The following parallelizing techniques are adopted:

- Inter-layer Parallelism

In inter-layer parallelism, the accelerator has a feed-forward hierarchical structure that can process a succession of data-dependent layers. They are executed in a pipelined fashion by executing a layer while preparing the next layer data to be processed. In this way, the accelerator utilized area is decreased significantly, which makes it easy to fit it on FPGAs or develop a low foot-print chip.

- Intra-Feature Map Parallelism

In intra-feature map parallelism, a group of output feature map pixels of a single output feature map plane are processed in parallel, which reduces the required processing time by acceleration factor x. This depends on output feature map and kernel sizes.

- Intra-convolution Parallelism

The last adopted parallelism is the intra-convolution, in which the processing of 2D convolution layers are implemented in a pipelined/parallel fashion.

Table 4.1: Required #PEs per kernel

Kernel size	#PEs/kernel	Convolution opcode
7x7 convolution	49	00
3x3 convolution	9	01
5x5 convolution	25	10
1x1 convolution	1	11

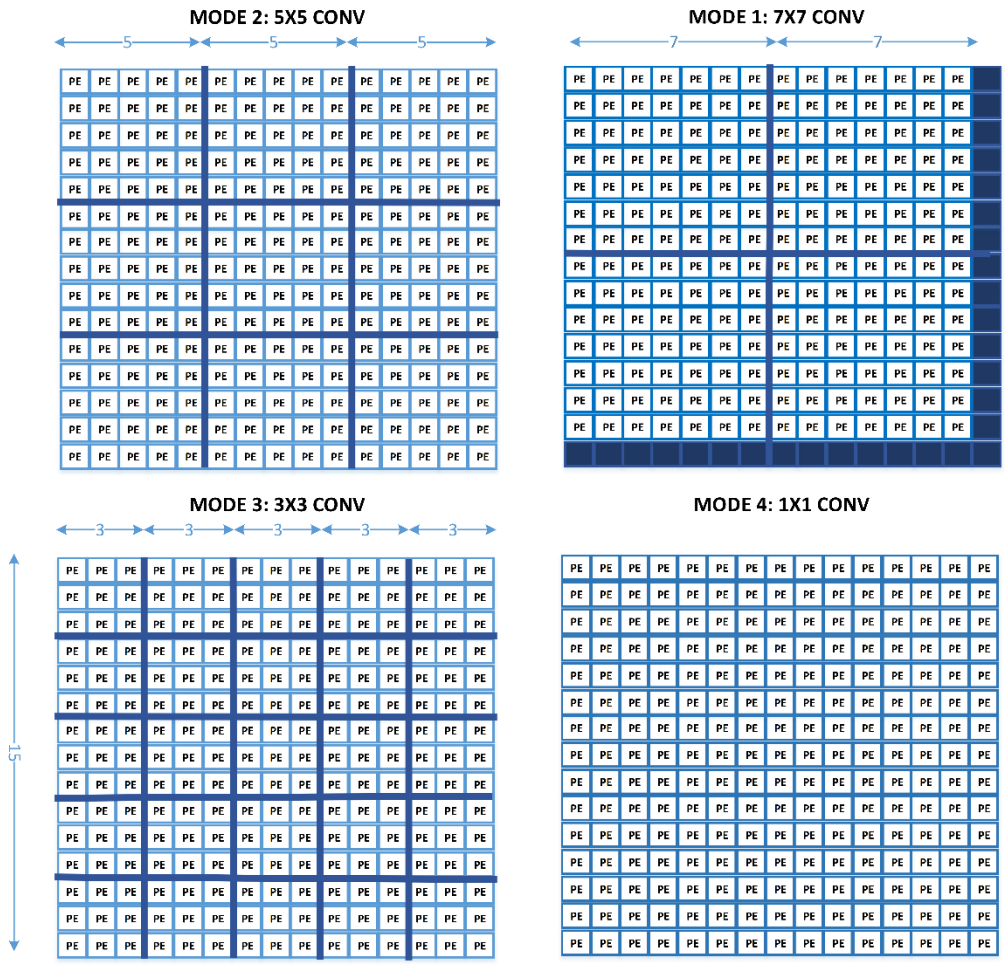


Figure 4.2: Different applied kernel sizes on PEs

There is a trade-off in the selection of the suitable number of parallel elements (PEs) between the acceleration factor and the accelerator size. Firstly, an analysis of GoogLeNet CNN layers is made to determine the suitable PEs count. GoogLeNet has four different convolution kernel sizes, which are 1x1, 3x3, 5x5 and, 7x7 kernels. The convolution opcode is represented by two bits to select the convolution type in different blocks by the control unit as listed in Table 4.1. In addition, the number of PEs is chosen to be 224 PEs that processes 224 kernels of 1x1conv

4.2. Loop Tiling

The capacity of buffers in FPGAs is not large enough to store all weights and intermediate feature maps (FMs) of all CNN layers. Consequently, loop-tiling is used to fetch the upcoming parts of feature maps in addition to kernel weights while processing the currently loaded ones. Feature maps and kernels of convolution layers are batched in a way that kernel weights are loaded only once, and FM tile is loaded once per batch. This factorization is employed to increase the data reuse and computational throughput as well.

Convolution layer pseudo-code for one layer is shown in Figure 4.3, which consists of nested for-loops. The first two for-loop iterate over the output feature map rows and

columns. The U for-loop iterates over the output channels. Also, the for-loops of V and iterates over input channels. Finally, the last two for-loops iterate over kernel rows and columns.

Some loops are selected to be unrolled to speed up the processing and parallelize the processing of certain iterations on the hardware. The number of parallelized iterations is called the unroll factor. Selecting suitable unroll factors might lead to huge hardware utilization. For the proposed processor, the for-loops of rows and columns are completely unrolled. Moreover, the for-loops of feature map rows and columns are tiled with a size of feature map row. The tile is reused by shifting the rows up by the stride value and other rows are reused again. Finally, the output channel is parallelized by processing multiple kernels and writing out multiple output pixels in parallel. In addition, Figure 4.4 shows an example of input feature map (IFMAP) tile to PEs fetching for 5x5 convolution. Parallel FIFOs reads a tile of five rows of IFMAP to start 5x5 convolution on PE cores. They are reused for multiple iterations, then FIFOs shifts up all the rows and read a new row, then repeats the operation again and so on.

```
for (n=0; n<N; n++) {                    //output FM rows
 for (m=0; m<M; m++) {                   //output FM columns
  for (u=0; u<U; u++) {                  //output channels
   for (v=0; v<V; v++) {                 //input channels
    for (i=0; i<K; i++) {                //kernel rows
     for (j =0; j<K; j++) {              //kernel columns
        Fout[u][n][m]+=Fin[v][S*n+i][S*m+j]*K[u][v][i][j]
  } } } } }
        Fout[u][n][m]+=bias[u]
}
```

Figure 4.3: Convolution layer pseudo code

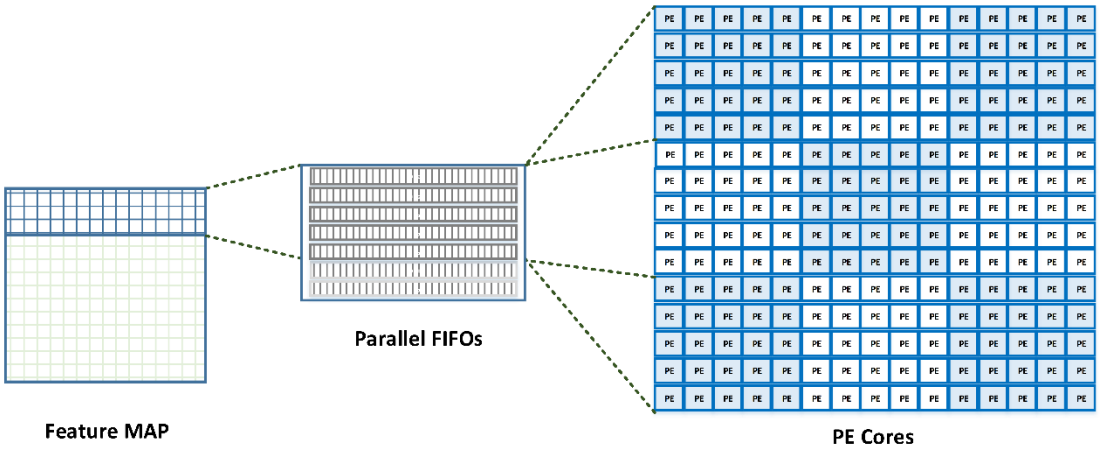


Figure 4.4: IFMAP tile to PEs – 5x5 convolution example

4.3. Memory Organization

Memory organization is one of the main challenges during accelerator design. As discussed before, memory access requires careful handling and planning. The final memory organization is set after analyzing several options and selecting the best implementation. Firstly, the limited number of access ports of memory is overcome by dividing the memory into 256 banks to read/write in parallel. Secondly, adding multiple buffers resolves the stalls due to memory dependencies and fetching cycles. While the proposed accelerator is built in a pipelined fashion, separate memories for weights and temporary data are used. The proposed architecture consists of multiple hierarchy levels of storage as follows:

- It consists of 256 Memory banks to save the partial summations during computations. They are implemented in FPGA BRAMs.
- Weights memory saves all weights of the CNN model. It utilizes 3Mb and are implemented in FPGA BRAMs.
- Weights Masks memory saves all weight masks. If the weight is a non-zero value, its value is fetched from weights memory.
- Weights buffer fetches the weights from weights memory and prepare it for parallel fetching to processing unit.
- IFMAPs buffer loads the feature maps from Memory banks and prepare them for FIFOs.
- Seven parallel FIFOs load complete seven rows from the input buffer. They store it while convoluting them with filter kernels.
- The internal register in each PE saves the loaded weight till the processing unit (PU) finishes.

This mechanism results in high data reuse because it enables global fetching for all loaded kernels with the same loaded feature map part on FIFOs. In addition, it empties the input buffer to be able to load more IFMAPs. This mechanism is designed by considering the latency of buffer loading to illuminate any stalls during convolution. As it loads more values than the needed next row of IFMAP while convoluting the loaded rows on FIFOs except 7x7 convolution as it has a stride with two, which shifts out two rows every shifting up. Moreover, memory bottleneck is one of the two main challenges that face the design of hardware accelerators. This is resolved by using 256 memory banks, and the processing unit became able to accumulate the partial sums of all needed convolution sizes without the usage of intermediate buffers.

4.4. Weights Decompressing

Weights memory saves the weights with 12-bit word length. After performing quantization on all weights, the weight's word length became 4-bit, which makes the memory be able to store 3x weights more than before. The weights buffer prepares the weights for parallel shifting to the processing unit based on current convolution layer sizes. After applying weights pruning on GoogLeNet model, the non-zero weights are reduced from ~6.9 million to ~0.96 million weights. This reduction makes it possible to use only FPGA BRAMs for weights storage, but memory decompression becomes a mandatory step to decompress 0.96M to 6.9M during computations. Weights masks are

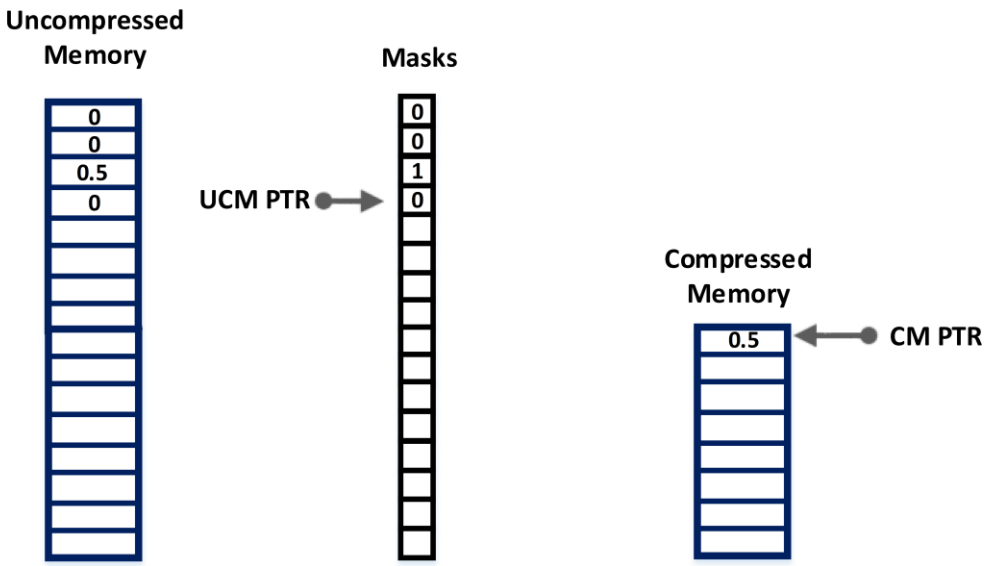


Figure 4.5: Masks map and weights decompressing

stored in the memory and they are checked each time to add a zero or non-zero value. This operation is showed clearly in Figure 4.5 by a chart of the process. The weights fetching scenario goes as follows, the weight control unit (WCU) checks the next bit mask. If the bit is 0, it writes a zero in the weights buffer. If it is equal to 1, the WCU reads the weight value from the weights memory and writes it into the weights buffer. The design is verified against any stalls because of weights fetching delays as the next weights become ready while the processing unit is running the currently loaded weights.

4.5. Processing Unit

The processing unit (PU) consists of 224 parallel elements, summation unit, bias unit and PU control unit. The parallel element consists of one multiplier in addition to two multiplexers as shown in Figure 4.6. The first multiplexer for weight input that selects between the stored weight or a new value. The second MUX selects between different FIFO fetched elements locations based on convolution kernel sizes, such as 1x1conv, 3x3conv, 5x5conv, and 7x7conv. The multiplier is built with simply shifting right block as all input weights are quantized to multiple of 2's number, less than one.

The summation unit is built of hierarchal adders to reduce the number of adders for different convolution sizes. This is resolved by using 24 adders with 9-inputs only instead of many adders with different input sizes.

The data flow while performing convolution computations is made with the proposed mechanism to increase the data reuse. The input feature map is stored in the input buffer, which in turn fetches the parallel FIFOs for every convolution patch. The parallel FIFOs slide the convolution subregion to the PE cores every cycle till reaching the end of row. Figure 4.7 shows and an example of 7x7 convolution where the parallel FIFOs are fully used. As discussed before, the 7x7 convolution is accelerated by 4x times by running 4 kernels at the same time. Figure 4.7 (a) shows the first write for 4 output feature maps. It is the first partial sum to be written which is fetched and added

with the new partial value. Secondly, Figure 4.7 (b) is the second write after shifting by stride value with 2. The third case is in Figure 4.7 (c) where the convolution for the currently loaded rows is finished, so the parallel FIFOs shift up the old two rows by newer two rows, then load the last two rows. After shifting up, the FIFOs work usually as shown in Figure 4.7 (a) and (b) by writing the partial sums to output feature maps, but in the next rows.

4.6. Control Units

The controlling of the system is made based on eight distributed-hierarchal control units (CUs) in addition to the main CU to simplify the controlling for each unit. Every CU is controlled by the main CU. On the other hand, every unit's CU controls all unit's related signals. Every CU is designed with a finite state machine that acts based on the stored values in their memories. This makes it easier to adopt and run other CNNs by changing the CU RAM values. The CUs are as follows:

- Main CU
- Processing unit CU
- Partial sums accumulation CU
- IFMAP fetching CU
- Fully connected CU
- Maxpooling CU
- Average pooling CU
- Softmax CU

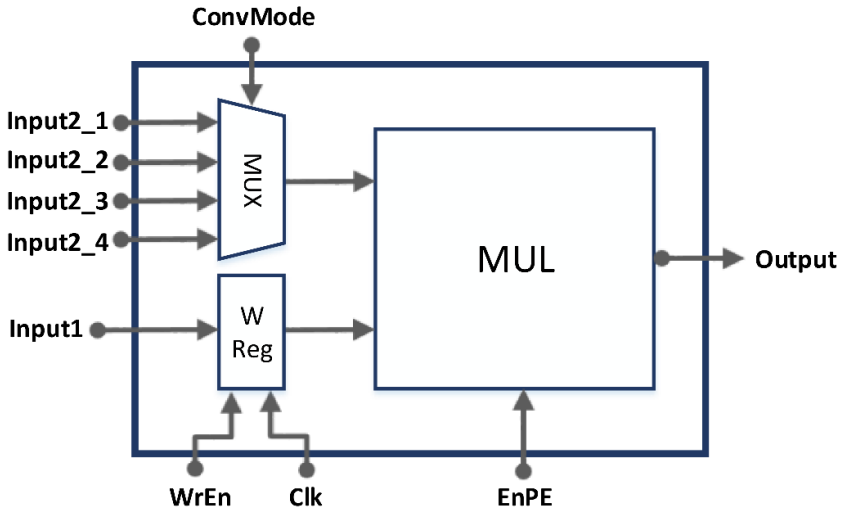


Figure 4.6: Parallel Element structure

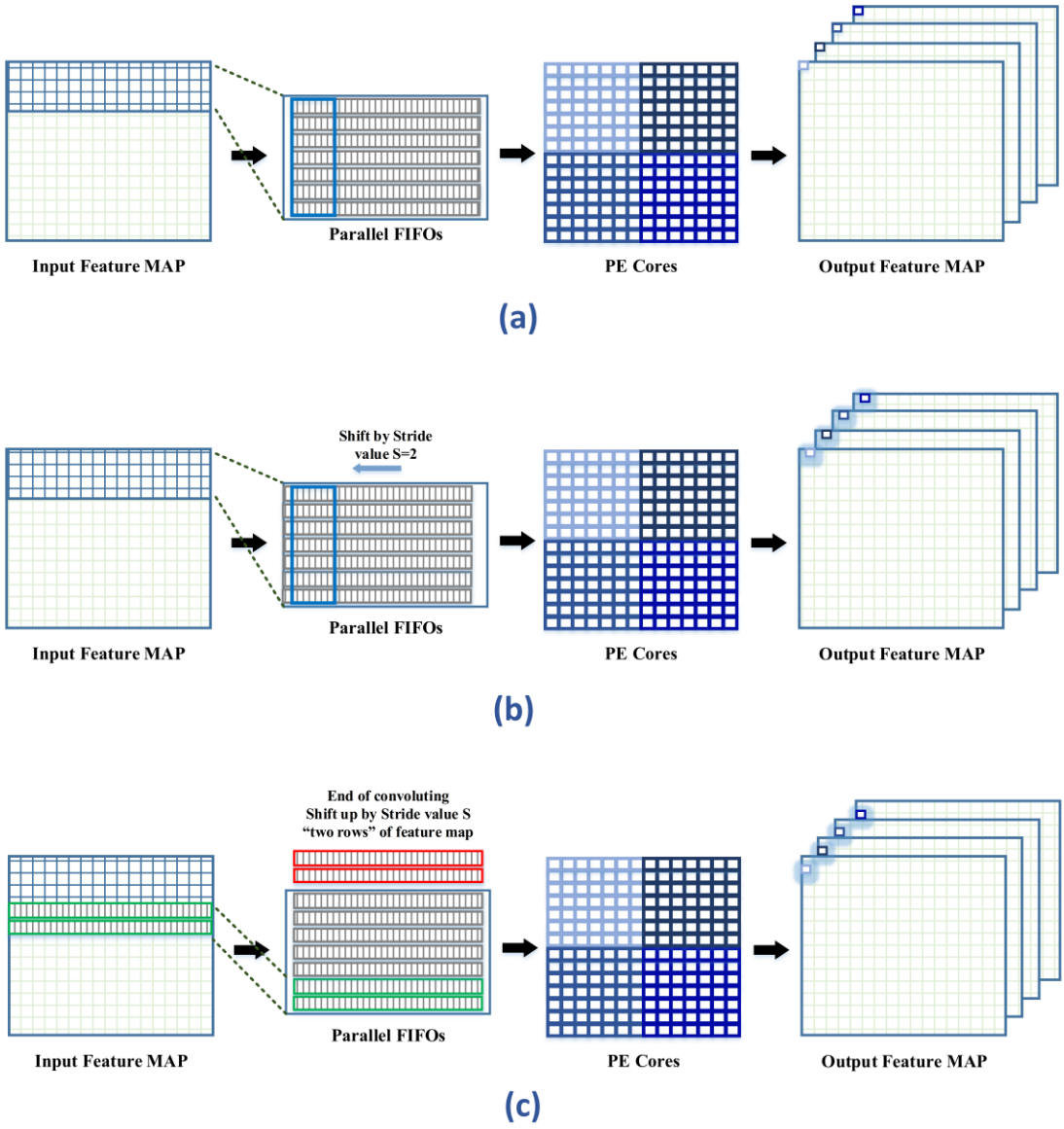


Figure 4.7: 7x7 Convolution example with parallel FIFOs and PE cores

4.7. Fully Connected Unit

As discussed before, FC layers are memory-centric. They usually contain millions of weights, and each weight is used only once. Since each weight in FC layers is used in one inference process only, it leaves no chance for reuse. The limited bandwidth degrades the performance significantly as loading those weights might take a long time, so it requires a careful design for this unit.

Firstly, a fast analysis of FC is presented. (4.1a) and (4.1b) represent a pseudo code for the FC layer. The output of average pooling is 1024 activations, which is the N value. It is stored in an intermediate buffer as input activations for FC, then it is fetched to PU. The network is trained on the ImageNet dataset with 1000 classes, so the M is equal 1000, which is the FC output.

$$out_m = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} W_{mn} x A_n + B_m \quad (4.1a)$$

for (m=0; m <M; m++) {
for (n=0; n < N; j++) {
$out_m \mathrel{+}= W_{mn}\ x\ A_n$ }
$out_m \mathrel{+}= B_m$ (4.1b)

4.7.1. FC Memory Management.

Fully connected processing requires 256 weights every cycle in the proposed design, which is not valid if they are fetched from weights memory directly. After performing memory compression as discussed in memory compression chapter, a lot of weights are suppressed to zero after weights pruning specially in the fully connected layer. An analysis is made on fully connected weights to discover their weights values. It is found that the number of non-zero weights per 256-tile does not exceed 32 weights. This make it easy to decompress 256 weights per cycle while knowing that there are 32 non-zero weights by maximum. The decompression unit is implemented and integrated with weights unit to use it while FC processing without any memory stalls. Moreover, the input activations are fetched tile by tile with 256 tile size to PEs and used for 1000 cycles before fetching the next tile. This leads to high data reuse for activations instead of read/write them every cycle.

4.7.2. FC Computation Management

The parallel elements are used for FC multiplications with extra 32 shifting blocks to make full use of the processing unit. The acceleration of FC is made for the inner loop by processing a tile of 256 weight each cycle. Therefore, the inner loop is processed in 4 cycles instead of 1024. Consequently, the fully connected layer is accelerated by 256x than a single MAC unit. The tilling diagram is shown in Figure 4.8. The diagram illustrates the process of adopted FC computation. The flow is as follows:

1. Every cycle, new 256 weights are fetched to PU.
2. An input activation tile is updated every 1000 cycles.
3. The multiplication is performed and forwarded to a parallel adder with 256 inputs. Finally, the adder's output is saved to the output register file.
4. After the first inner loop of pseudo code in (1b) is completed, the output of every summation is added to its corresponding value in the output register file, and so on till finishing all tiles.

4.8. Maxpooling Unit

Maxpooling is used between convolution layers to reduce the spatial size of feature maps. There are 14 Maxpooling layers in GoogLeNet. Maxpooling unit works on four feature maps in parallel. Every unit consists of an input buffer, output buffer, and comparators.

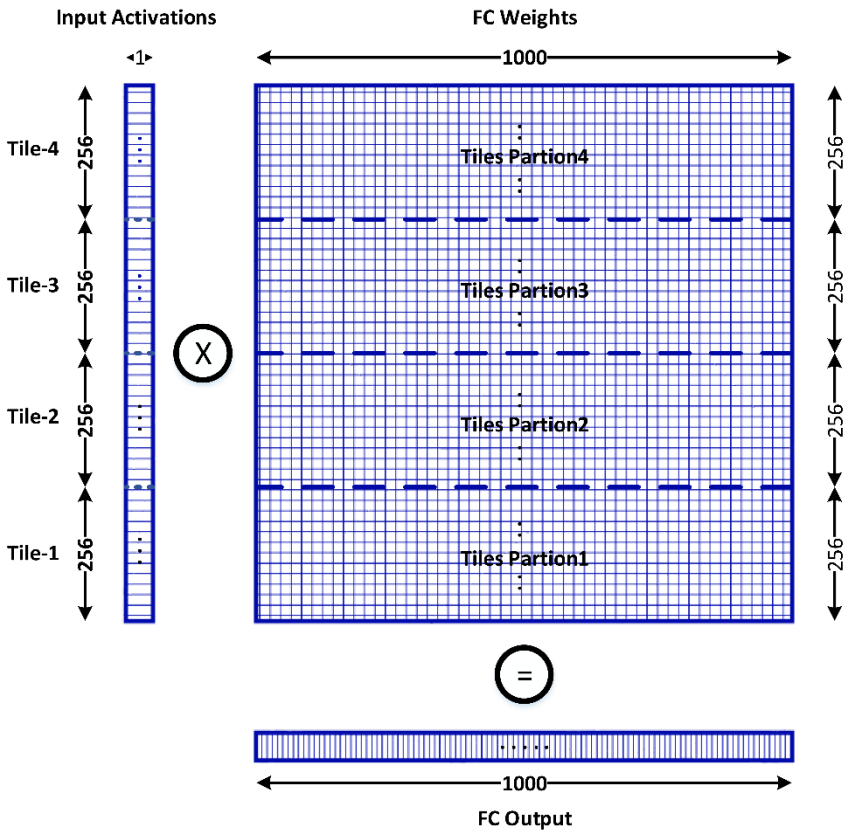


Figure 4.8: Fully Connected layer tiling diagram

At first, the data flow for Maxpooling worked by fetching a complete three rows from every input feature map into the input buffer, then start sliding the subregion into a 9-input comparator to get the final output in the output buffer. This way makes the input buffer have a larger size in addition being busy for long time. Moreover, building a 9-input comparator is an extra cost for utilized area.

The second data flow reads every input feature, row by row. This makes it easy to build a smaller input buffer and start working immediately with input data. Also, the comparator becomes smaller with 3-input only. The extra overhead is handling the output buffer data as the written result every cycle is kept till completing all comparisons. Firstly, the first comparator output is written as shown in case (a) of Figure 4.9. Secondly, the next output of the second row is compared with previously written value, then write the max as shown in the case (b) of Figure 4.9. Finally, the next output of third row is compared with the written one while writing the max as shown in case (c) of Figure 4.9. The final result becomes ready to be released and written to memory banks again. Figure 4.9 case (d) is an example of sliding Maxpooling subregion and writing a new output pixel to the output buffer. It is worth mentioning that, case (a) always followed by case (d) while processing. As case (b) for example comes while reading the second row of the input feature map. This solution is more optimum, so it is used for four parallel Maxpooling blocks.

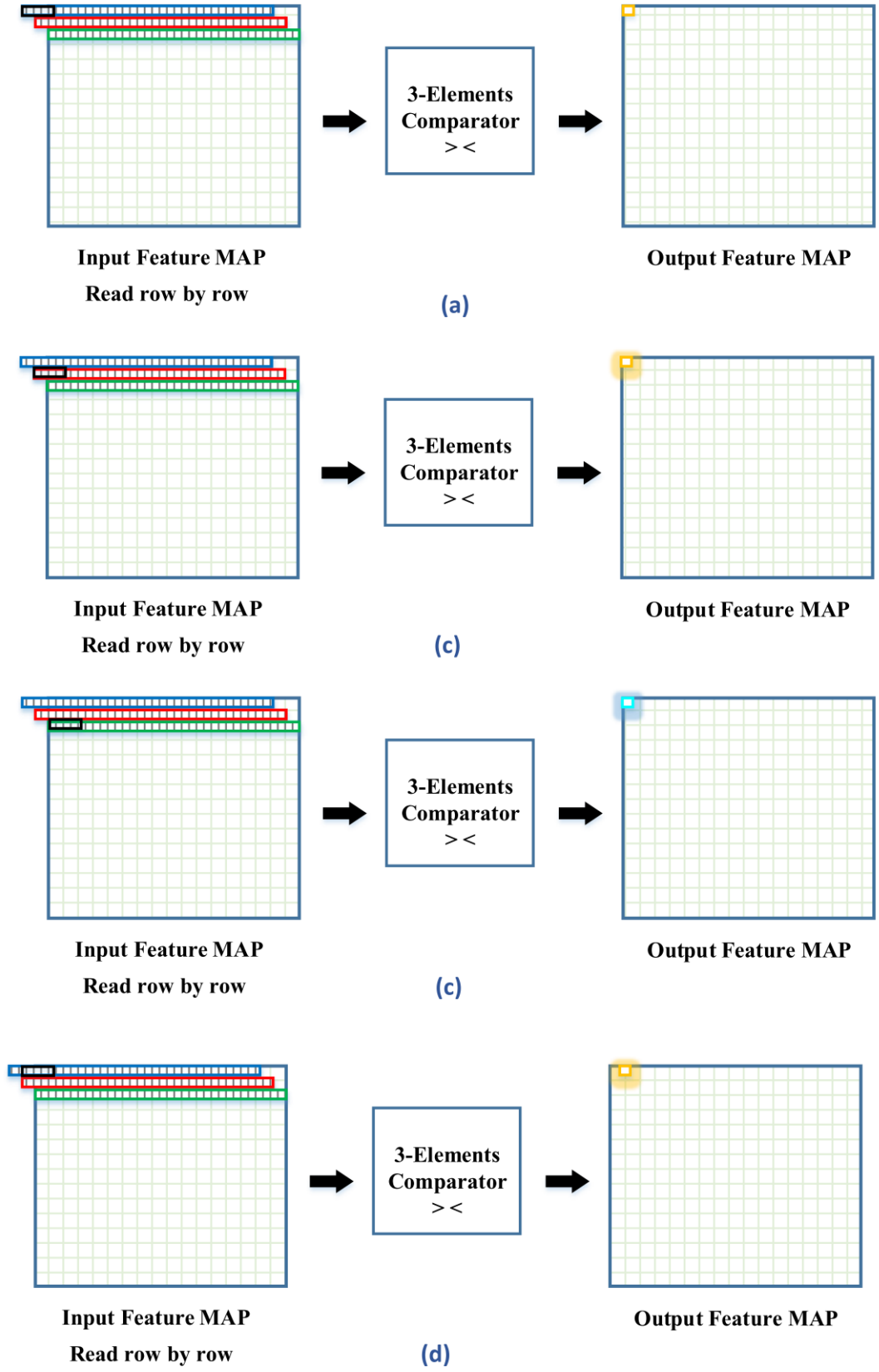


Figure 4.9: Maxpooling data flow. (a) First part of comparator output, (b) Second part of comparator output, (c) Third part of comparator output, and (d) first part of second pixel comparator output

4.9. Local Response Normalization Unit

Local response normalization (LRN) is used to normalize the distribution of the input activations by normalizing over local input regions. It depends on the activations of adjacent kernels at the same layer [41]. This is made instead of computing mean and deviation as performed by the batch normalization (BN) layer. LRN does not have any learnable parameters and all computations are made between input activations as shown in (4.2). The parameters (α, β, k, n) are set firstly γ =0.0001, k=1, β =0.75, and n=5. The parameter n represents the number of input activations $a_{x,y}^{j}$ that is squared and summed to compute the normalized activation. After investigating the LRN equation, it needs a lot of computations to generate normalized activations. Squaring, division, and powering blocks in addition to intermediate registers are needed, which takes up a large area and power consumption to compute it.

Instead of building these large blocks, a software experiment is done on the GoogLeNet model using the ImageNet testing set to get the average difference before and after the LRN layer. This average difference is computed across input channels and testing images. The average values are ranging from 0 to 0.006, which are added randomly across input channels instead of making all this computation. The overall accuracy does not affect as it is well known that the CNNs themselves add up noise through different layers. This is proven experimentally by replacing LRN with a randomizer using batches of testing images, every batch contains 128 images. The overall accuracy is ranging from 0.02:-0.02, or does not change in several testing batches. This is done by using the saved random values from the previous software experiment.

$$b_{x,y}^{i} = \frac{a_{x,y}^{i}}{(k+\alpha \,.\, \sum_{j=\max(0,i-\frac{n}{2})}^{\min(N-1,i+\frac{n}{2})} (a_{x,y}^{j})^{2})^{\beta}} \tag{4.2}$$

4.10. Average Pooling Unit

The average pooling layer is added before the fully connected layer to reduce the input feature map size to the fully connected layer to 1x1024 instead of 7x7x1024. It simply adds up all pixels of every 7x7 feature map and divides it by 49. The unit works on eight feature maps in parallel and stores the output in an intermediate buffer for the fully connected layer.

4.11. Softmax Unit

Softmax unit is used to convert the output of a fully connected layer to probability distributed values [40]. The unit consists of ten parallel CORDIC cores to compute the exponential function. The unit stores exponential outputs again in the buffer while computing their summation. After computing the summation of 1000 exponentials, every exponential is divided by the summation and stored in the final output buffer. As shown in (4.3), N is equal to 1000 as the number of classes is 1000 classes. The block diagram is shown in Figure 4.10.

$$f(i) = \sum_{i=0}^{N} \frac{e^{a_i}}{\sum_k e^{a_k}} \tag{4.3}$$

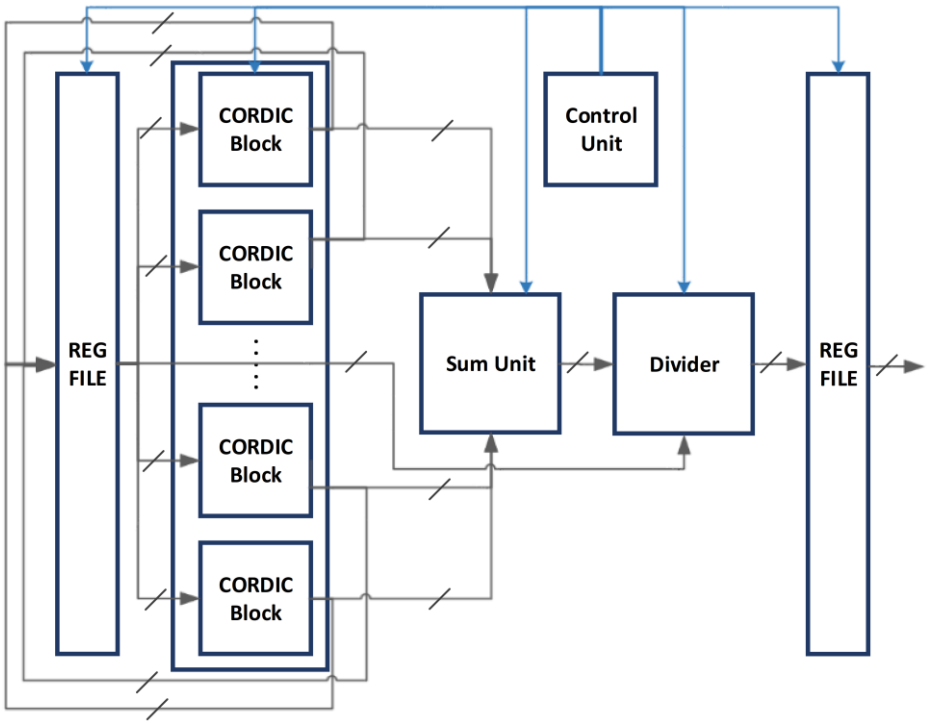


Figure 4.10: Softmax unit schematic

4.12. Processor Modifications

The DSP resources of the FPGA are firstly used to implement PEs multipliers, which increased the power consumption while processing convolution and fully connected layers. After memory compression and quantization, the weights are quantized to 4-bit only, and they become one of a few distinct values. As a result, the multiplication is made simply by shifting after decoding these weights based on the decoding table in Table 4.2. This modification lets the processor be DSP-free, and the power consumption of multipliers is saved as the conventional multipliers became a simple rewiring instead of large conventional adders.

Furthermore, convolution kernels with equal size are processed at the same time, which makes some of the parallel cores are unutilized during layers computations. This is resolved by enabling the processing of multiple kernel sizes in parallel, which increases the utilization of the cores. Finally, the time overhead for writing and reading all padding pixels is skipped to save these cycles. Consequently, nearly 240,000 cycles are saved for writing and thousands of cycles reading.

Table 4.2: Weights decoding table

Weight value	Decoded code	Shifting	Sign
0.5	0001	>>1	+ve
0.25	0010	>>2	+ve
0.125	0011	>>3	+ve
0.0625	0100	>>4	+ve
0.03125	0101	>>5	+ve
0.015625	0110	>>6	+ve
0.0078125	0111	>>7	+ve
-0.5	1001	>>1	-ve
-0.25	1010	>>2	-ve
-0.125	1011	>>3	-ve
-0.0625	1100	>>4	-ve
-0.03125	1101	>>5	-ve
-0.015625	1110	>>6	-ve
-0.0078125;	1111	>>7	-ve

Chapter 5 : Discussion and Results

In this chapter, the experiment of selecting the fixed-point precision is presented, and the theoretical throughput is computed. In addition, the resource utilization of the proposed processor is reported, and power consumption report by Vivado is presented. Then, a comparison is made between Intel Core-i7 CPU, NVidia GTX 1080Ti GPU, and the proposed accelerator by showing power consumption improvement. In addition, a comparison between the proposed work and popular AI embedded accelerators such as NVidia Jetson Nano and Intel Movidius is presented. Finally, a comparison between the existing GoogLeNet implementations and the proposed accelerator is provided.

5.1. Selecting Fixed-point Precision

The effect of word length is tiny on the accuracy of convolutional neural networks as stated in the literature [41-42]. 12-bit fixed-point arithmetic operators are used instead of 32-bit word-length to reduce storage size and power consumption during operations. Several experiments are held to select the suitable arithmetic operator width while keeping the accuracy loss at least. The experiments are done on an epoch of 1024 images from the ImageNet dataset to see the effect of sweeping the word length. The model is implemented in software by providing the maximum and minimum values that are represented by the accelerator, and every output activation of each layer is suppressed to zero or truncated to this word length. The first experiment is done to select the integer part width. Width of 4-bit is selected for integer part to be able to represent the maximum integer value, which keeps the accuracy without any loss. The second experiment is done for the fractional part while fixing the integer part at 4-bit. The second experiment loss is depending on the width of the fractional part.

Figure 5.1 shows the accuracy loss for top-1, top-3, and top-5 losses when using 16-bit to 9-bit fixed-points. The number of bits represents the whole word length. For example, at 14-bit word length, the fractional part is 10-bit. The usage of 8-bit word length gave the worst accuracy with a loss of nearly 30%. By increasing the length gradually, the loss started to decrease to zero accuracy loss at 15-16 bits. It is observed that using 12-bit width with giving 8-bit to the fractional part gives an accuracy loss with 0.01 while keeping it multiple of 4's. The word length reduction experiment is done on the inference phase only to use it by the accelerator, so while training, all values are represented by full precision.

5.2. Theoretical Throughput

On the other hand, theoretical throughput is calculated to compare it with the actual throughput. Theoretical throughput is calculated as follows:

#convolution cycles/frame=

$$\sum_{l=1}^{L} \frac{\text{kernel size}}{\#parallel\ kernels}\ x\ OFMAP\ Size\ x\ \#IFMaps \tag{5.1}$$

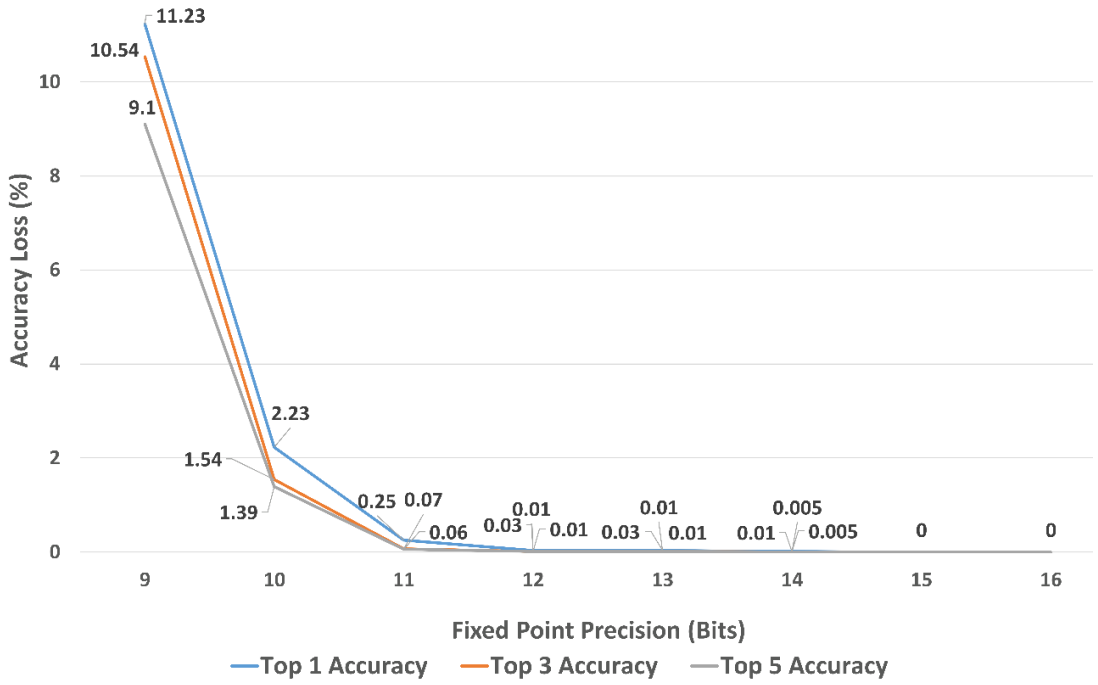


Figure 5.1: Accuracy loss versus fixed point precision

(5.1) calculates the needed number of cycles for all 57 GoogLeNet convolution layers. Also, this count is added to the needed cycles for Maxpooling, average pooling, fully connected, and Softmax layers processing. The overall theoretical throughput is 30.3fps at a frequency of 200MHz.

5.3. Design Testing

Design testing is an important step to validate the design functionality. Firstly, testing for each independent unit are done by testing the unit with critical cases to resolve any issue. The testing for each unit is done interactively to trace every signal and try different inputs. The integration is performed gradually between the design units as shown in Appendix C for the hardware accelerator units' organization. Also, interactive testing is done in every step. After integrating all units, a top-level test bench is used to test and validate the proposed hardware accelerator. Testing images are converted earlier to binary RGB format and written in separate files using Matlab. The processing is enabled by the "Start_CNN" signal after the "reset" signal goes down as shown in Figure 5.2. The accelerator keeps running till finishing all layers, which is identified by getting "ProcessorDone" signal high. Softmax layer is computed using softmax unit, which runs till getting the highest class probability in addition to its class number. "ClassPrediction" and "ClassNumber" become available after getting "SoftMaxDone" signal high as shown in Figure 5.3. "ClassPrediction" is 12-bit fixed point variable. As the class prediction is a probabilistic value, 1 bit is set for sign, 1 bit for integer part, and 10 bits are set as fractional part. Moreover, "ClassNumber" is a 10-bit integer variable that represent the class number from 0 to 999. Finally, the class number is mapped to its class name on Matlab. Appendix D shows an example for Image classification on the proposed hardware accelerator and the result is compared with the software result on python model.

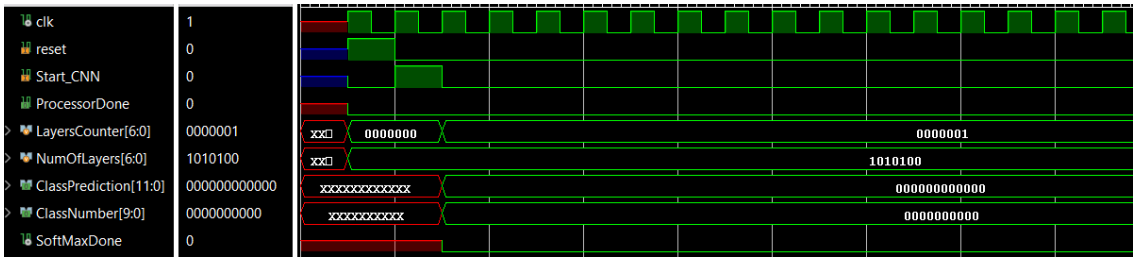


Figure 5.2: Top-level signals at the start of processing

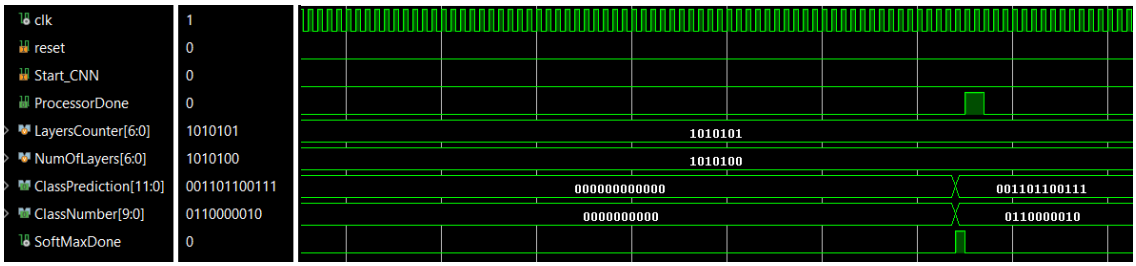


Figure 5.3: Top-level signals at the end of processing

5.4. Area Utilization and Power Consumption

The proposed accelerator is implemented in native RTL (Verilog) on Virtex-7 FPGA. Xilinx Virtex-7 FPGA VC709 is a popular FPGA kit which is widely used in high performance applications as presented in Appendix E. It is chosen due to several reasons. Firstly, it has a huge number of logic cells of 693,120 cells. Secondly, memory resources are important factor for selecting it with 52,920 Kb. There are 3600 DSP slices, but they do not be used after converting all conventional multiplications into shifting operations. There are other FPGA boards that have more resources, but with higher price which are not available.

Table 5.1 shows the system utilization on the FPGA. Thanks to weights quantization and compression, the accelerator is built with zero DSP units and on-chip BRAMs only. Power consumption is one of the main factors that qualify digital designs. Also, the design flavor for the proposed architecture is getting the best power consumption. This flavor is followed in any trade-off during the implementation. All optimization and approximation techniques invoked during this work have a huge impact on the final power consumption.

Table 5.1: The proposed hardware accelerator utilization on Virtex-7 FPGA

Resource	DSP	BRAM	LUT	FF
Used	0	1134	407290	85927
Available	3600	1470	433200	866400
Utilization	0%	77%	94%	10%

Full simulation is made on Vivado to generate a SAIF file, which is used to report an accurate power consumption without any estimations from the tool. SAIF stands for Switching Activity Interchange Format. The SAIF file saves information about toggle rates and static probability. After generating the SAIF file, it is imported in Vivado after synthesis to be used for power reporting. The report is generated under default conditions. As shown in Figure 5.4, the total on-chip power is nearly 3.9W with 0.45W static power consumption and 3.47W dynamic power consumption.

5.5. Comparisons

The proposed accelerator works on a maximum frequency of 200MHz. A comparison is made between Intel Core-i7 CPU, NVidia GTX 1080Ti GPU, and the proposed accelerator. The comparison is made in terms of the operating frequency, process technology, power consumption, performance (fps), and power efficiency, as shown in Table 5.2. The results show that the proposed accelerator provides the best performance in terms of the number of frames per Watt. The normalized power efficiency is 6.4 frames/Watt for the proposed accelerator, 0.81 frames/Watt for NVidia GTX 1080Ti GPU, and 0.128 frames/Watt for Intel Core-i7. It is worth mentioning that the used FPGA is fabricated with 28nm technology, which consumes a power more than 14nm and 22nm technologies. The proposed accelerator has 49.5x improvement over Intel Core-i7 and 7.8x over NVidia GTX 1080Ti.

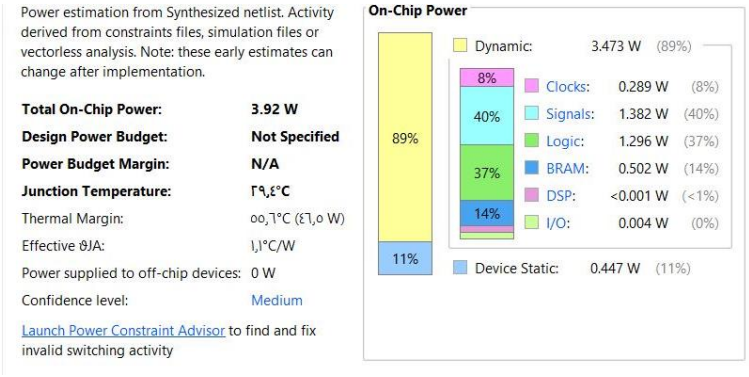


Figure 5.4: Power report by Vivado using a generated SAIF file

Table 5.2: Comparison with other platforms

	Intel Core-i7	**NVidia GTX 1080Ti**	**This work**
Clock	3.1 GHz	1.5 GHz	200 MHz
Technology	22nm	16nm	28nm
Power (W)	15	106	3.92
Performance (fps/s)	1.92	85.83	25.1
Performance[(1)] (fps/s)	0.124	11.45	25.1
Power Efficiency (fps/W)	0.128	0.81	**6.4**

(1) Normalized performance to 200 MHz frequency

The developers have started to use embedded AI accelerators for deploying their deep learning applications. NVidia Jetson Nano and Intel Movidius NCS are from these popular accelerators. Table 5.3 shows the comparison between the proposed hardware accelerator, NVidia Jetson Nano, and Intel Movidius. Firstly, Jetson Nano is used to run GoogLeNet using two frameworks: Caffe and TensorRT at a frequency of 920MHz [49]. Caffe framework is widely used in deep learning development, while TensorRT framework is developed by NVidia to accelerate the inference process. Secondly, Intel Movidius NCS (Neural Compute Stick) runs GoogLeNet using Caffe framework at a frequency of 933MHz [50]. All inference experiments are done with batch size 1. The table shows that the proposed hardware accelerator overcomes Jetson Nano and Intel Movidius while running with Caffe framework, but Jetson has a better performance using TensorRT framework. The power consumption is 5W for Jetson and 3.92W for the proposed design. Unfortunately, Intel Movidius power consumption for GoogLeNet is not mentioned in the experiment. The power efficiency is the best for Jetson Nano using TensorRT framework with 12 frames/Watt, but the proposed implementation is better while using Caffe framework with 6.4 frames/Watt.

Another comparison is made between the proposed accelerator and GoogLeNet hardware accelerators in the literature, as shown in Table 5.4. The first implementation is Zhao's hardware accelerator, which is an ASIC chip built with 65nm technology. The second implementation is Gokhale's FPGA implementation on Zynq XC7Z045. The third implementation is CoNNA_C3 on Zynq ZCU102. The comparison is made between the implementations in terms of the operating frequency, fixed-point precision, process technology, power consumption, performance, and power efficiency, as shown in Table 5.4. The results show that the proposed accelerator provides the best performance in terms of the number of frames per Watt. In addition, it overcomes Gokhale's implementation in terms of peak performance and power consumption. Gokhale's implementation computes the number of frames per second for convolution layers only, so it processes 27.2fps compared to 25.1fps for the proposed implementation. In addition, Zhao's implementation overcomes the proposed accelerator in terms of GOP/s as it works on 650MHz. Also, it is an ASIC implementation, so the power consumption is expected to be lower than the FPGA implementations. Also, the proposed hardware accelerators overcomes the performance of CoNNA_C3. The power consumption is not mentioned for CoNNA_C3 implementation, but it's expected to be higher than the proposed implementation as it uses offline DRAMs and ARM processor with Zynq FPGA. While the other

Table 5.3: Comparison with popular embedded AI accelerators

	NVidia Jetson Nano [51]	**NVidia Jetson Nano [51]**	**Intel Movidius NCS [52]**	**This work**
Framework	Caffe	TensorRT	Caffe	-
Frequency	920MHz	920MHz	933MHz	200MHz
Power (W)	5	5	-	3.92
Performance (fps/s)	19	60	13.66	25.1
Performance[(1)] (fps/s)	4.13	13.1	2.93	25.1
Power Efficiency (fps/W)	3.8	12	-	6.4

[(1)] Normalized performance to 200 MHz frequency

implementations use a plain GoogLeNet CNN model, the proposed implementation uses a compressed CNN model. This is one of the design advantages which improves the power consumption as discussed earlier.

The data access patterns variations in CNNs make it difficult for custom architectures to get higher utilization efficiency while processing all CNN layers. Consequently, utilization efficiency is stated as the ratio of the actual number of operations processed to the theoretical maximum number of the processed operations. This is translated to the ratio of actual fps to the theoretical fps for a given CNN. Table 5.4 shows that the utilization efficiency is 83% for Zhao's work, 91% for Gokhale's work, and 89% for this work. Also, the work of Zhao and Gokhale computes the number of frames per second for convolution layer acceleration only, which is degraded when FC and average pooling layers are added.

Table 5.4: Comparison with other GoogLeNet hardware accelerators

	Zhao [19]	**Gokhale [18]**	**CoNNA_C3 [53]**	**This Work**
Platform	ASIC TSMC	Zynq XC7Z045	Zynq ZCU102	Virtex-7 VC709
Max Clock (MHz)	650	250	100	200
Precision	16-bit fixed	16-bit fixed	16-bit fixed	12-bit fixed
Process Technology	65nm	28nm	16nm	28nm
Power	859mW	9.48W	-	3.92W
Peak Performance (GOP/s)	242.4	116.5	17.325	129.2
Power Efficiency (GOP/W)	282	12.3	-	32.7
Power Efficiency (fps/W)	-	2.87	-	6.4
Performance(1) (fps)	23.6	27.2	4.95	25.1
Utilization efficiency	83%	91%	-	89%

(1) Normalized performance to 200 MHz frequency

Conclusion

Contributions

In this thesis, a power-efficient convolutional neural networks accelerator based on GoogLeNet CNN was proposed. Weights pruning and quantization were applied, which reduced the memory size by 57.6x with a top-5 accuracy loss of 2.6%. As a result, only FPGA BRAMs were used for weights and activations storage without using offline DRAMs. The compression model was explained in details, and the reduction for every GoogLeNet layer was presented. In addition, this accelerator used zero DSP units as it replaced all multiplications by shifting operations.

The hardware accelerator was built based on a time-sharing/pipelined architecture that could process the CNN model layer by layer. The architecture proposed a new data fetching mechanism that increased data reuse. Moreover, it used only 224 PEs. All accelerator units were implemented in native RTL (Verilog), and all control units could be reconfigured to process other CNNs successfully. Several optimization and approximation techniques were adopted to improve the design with a little loss in the accuracy.

Moreover, several improvements were applied lately, such as increasing cores utilization or skip padding cycles. A word length of 12-bit was used after performing several experiments to select a suitable word length. The proposed hardware accelerator classified 25.1 fps for GoogLeNet inference using 3.92W with a power-efficiency improvement more than the previous FPGA implementations for GoogLeNet. It provided 49.5x power-efficiency improvement over Intel Core-i7 and 7.8x over NVidia GTX 1080Ti. On the other hand, the proposed design processed the fully connected layer with 256x more than a single MAC unit. The proposed hardware accelerator achieved a top-5 classification accuracy of 91%, which was significantly higher than comparable architectures.

Future Work

Regarding future work, the control units in the proposed hardware accelerator can be reconfigured to process other CNN models such as ResNet or SqueezeNet. Also, the memory compression framework can be applied on the new CNN software model to get the proposed design benefits. In addition, the ASIC implementation can be made to get better performance in terms of power consumption, processing speed, and utilized area.

List of Publications

1. A. J. El-maksoud, M. Ebbed, A. H. Khalil, and H. Mostafa, “Power Efficient Design of High-Performance Convolutional Neural Networks Hardware Accelerator on FPGA: A Case Study with GoogLeNet,” in *IEEE Access*, vol. 9, pp. 151897-151911, 2021.

2. A. J. El-maksoud, et al., “FPGA Design of High-Speed Convolutional Neural Network Hardware Accelerator,” *in 3rd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, pp. 376-379, 2021.

References

1. A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Advances in Neural Information Processing Systems (NIPS)*, vol. 25, no. 2, 2012.
2. K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *Proc. Int. Conf. Learn. Representations*, 2015.
3. J. Mutch and D. G. Lowe, “Multiclass object recognition with sparse localized features,” *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, pp. 11-18, 2006.
4. O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, and M. Bernstein, “Imagenet large scale visual recognition challenge,” *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.
5. K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.
6. Y. Lee, H. Kim, E. Park, B. Yim, and H. Kim, “Optimization for object detector using deep residual network on embedded board,” in *IEEE Int. Conf. Consum. Electron.*, pp. 0–3, 2016.
7. R. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 580–587, 2014.
8. S. Tamura et al., “Audio-visual speech recognition using deep bottleneck features and high-performance lipreading,” *Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA)*, pp. 575-582, 2015.
9. S. Ramos, S. Gehrig, P. Pinggera, U. Franke and C. Rother, “Detecting unexpected obstacles for self-driving cars: Fusing deep learning and geometric modeling,” *IEEE Intelligent Vehicles Symposium (IV)*, pp. 1025-1032, 2017.
10. B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, et al., “An empirical evaluation of deep learning on highway driving,” *arXiv:1504.01716*, Apr. 2015.
11. A. Khan, A. Sohail, U. Zahoora and A. S. Qureshi, “A survey of the recent architectures of deep convolutional neural networks,” *arXiv:1901.06032*, Jan. 2019.
12. J. Hoffmann, O. Navarro, K. Florian, B. Janßen, and H. Michael, “A survey on CNN and RNN implementations,” *IARIA*, no. c, pp. 33–39, 2017.
13. E. Nurvitadhi et al., “Can FPGAs beat GPUs in accelerating next-generation deep neural networks?” in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, pp. 5-14, 2017.
14. K. Abdelouahab, M. Pelcat, J. Serot and F. Berry, “Accelerating CNN inference on FPGAs: A survey,” *arXiv:1806.01683*, 2018.
15. A. Shawahna, S. M. Sait and A. El-Maleh, “FPGA-based accelerators of deep learning networks for learning and classification: A review,” *IEEE Access*, vol. 7, pp. 7823-7859, 2019.
16. K. Guo, S. Zeng, J. Yu, Y. Wang and H. Yang, “A survey of FPGA-based neural network inference accelerator,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, no. 1, pp. 1-26, 2019.

17. C. Szegedy et al., “Going deeper with convolutions,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 1-9, 2015.
18. V. Gokhale, A. Zaidy, A. X. M. Chang and E. Culurciello, “Snowflake: An efficient hardware accelerator for convolutional neural networks,” in *IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1-4, 2017.
19. Zhao, B., Li, J., Pan, H., and Wang, “A high-performance reconfigurable accelerator for convolutional neural networks,” in *ICMSSP*, pp. 150–155, 2018.
20. 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,” in *Proc. 19th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, pp. 269–284, 2014.
21. Y. Chen, T. 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, June 2019.
22. U. Aydonat, S. O’Connell, D. Capalija, A. C. Ling and G. R. Chiu, “An OpenCL deep learning accelerator on arria 10,” in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, pp. 55-64, 2017.
23. J. Zhang and J. Li, “Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network,” in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, pp. 25-34, 2017.
24. N. Suda et al., “Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks,” in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, pp. 16-25, 2016.
25. C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao and J. Cong, “Optimizing FPGA-based accelerator design for deep convolutional neural networks,” in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, pp. 161-170, 2015.
26. S. I. Venieris and C.-S. Bouganis, “FPGAConvNet: A framework for mapping convolutional neural networks on FPGA,” in *Proc. IEEE 24th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, pp. 40-47, May 2016.
27. Y. Umuroglu et al., “FINN: A framework for fast scalable binarized neural network inference,” in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, pp. 65-74, 2017.
28. S. Chakradhar, M. Sankaradas, V. Jakkula and S. Cadambi, “A dynamically configurable coprocessor for convolutional neural networks,” in *Proc. 37th Annu. Int. Symp. Comput. Archit. (ISCA)*, pp. 247-257, 2010.
29. M. Sankaradas et al., “A massively parallel coprocessor for convolutional neural networks,” *20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, pp. 53-60, 2009.
30. S. Han, J. Pool, J. Tran and W. Dally, “Learning both weights and connections for efficient neural network,” in *Proc. Adv. Neural Inf. Process. Syst.*, pp. 1135-1143, 2015.
31. Y. Gong, L. Liu, M. Yang and L. Bourdev, “Compressing deep convolutional networks using vector quantization,” *arXiv:1412.6115*, 2014.
32. D. A. Huffman, “A method for the construction of minimum-redundancy codes,” in *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098-1101, Sept. 1952.
33. Y. Guo, A. Yao and Y. Chen, “Dynamic network surgery for efficient DNNS,” in *Proc. Adv. Neural Inf. Process. Syst.*, pp. 1379-1387, 2016.
34. A. Zhou, A. Yao, Y. Guo, L. Xu and Y. Chen, “Incremental network quantization: Towards lossless CNNs with low-precision weights,” *arXiv:1702.03044*, 2017.

35. X. Gao, Y. Zhao, L. Dudziak, R. Mullins and C.-Z. Xu, “Dynamic channel pruning : Feature boosting and suppression,” in *Proc. Int. Conf. Learn. Represent.*, pp. 1-14, 2019.
36. S. Han, H. Mao and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning trained quantization and Huffman coding,” *arXiv:1510.00149*, 2015.
37. S. Mittal, “A survey of FPGA-based accelerators for convolutional neural networks,” *Neural Comput. Appl.*, vol. 32, no. 4, pp. 1109-1139, Feb. 2020.
38. K. Guo, S. Zeng, J. Yu, Y. Wang and H. Yang, “A survey of FPGA-based neural network inference accelerator,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, no. 1, pp. 1-26, 2019.
39. K. Lee, S. H. Sung, D. Kim and S. Park, “Verification of normalization effects through comparison of CNN models,” in *International Conference on Multimedia Analysis and Pattern Recognition (MAPR)*, pp. 1-5, 2019.
40. C. E. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, “Activation functions : Comparison of trends in practice and research for deep learning,” *arXiv:1811.03378*, 2018.
41. J. L. Holi and J. Hwang, “Finite precision error analysis of neural network hardware implementations,” *IEEE Transactions on Computers*, vol. 42, no. 3, pp. 281-290, March 1993.
42. D. Larkin, A. Kinane, and N. O’Connor, “Towards hardware acceleration of neuroevolution for multimedia processing applications on mobile devices,” *I. King, J. Wang, L. Chan, and D. L. Wang, editors, ICONIP (3)*, volume 4234 of Lecture Notes in Computer Science, pages 1178-1188, Springer, 2006.
43. M. Svensén and C.M. Bishop, “Pattern recognition and machine learning,” *Springer*, pp. 101, 2009.
44. X. Glorot, A. Bordes and Y. Bengio, “Deep Sparse Rectifier Neural Networks,” in *Proc. Conf. Artificial Intelligence and Statistics*, 2011.
45. M. C. Popescu, V. E. Balas, L. Perescu-Popescu and N. Mastorakis, “Multilayer perceptron and neural networks,” *WSEAS Transactions on Circuits and Systems*, vol. 8, no. 7, pp. 579-588, 2009.
46. M. A. Nielsen, “Neural Networks and Deep Learning, Determination Press,” 2015, [online] Available: http://neuralnetworksanddeeplearning.com/.
47. S. Khan, H. Rahmani, S. A. A. Shah and M. Bennamoun, “A guide to convolutional neural networks for computer vision,” *Synth. Lectures Comput. Vis.*, vol. 8, no. 1, pp. 1-207, 2018.
48. Y. Lu, “Artificial intelligence: A survey on evolution models applications and future trends,” *J. Manag. Analytics*, vol. 6, no. 1, pp. 1-29, 2019.
49. “NVIDIA Jetson Nano,” [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano, 2019.
50. “Intel Movidius VPU,” [Online]. Available: https://www.movidius.com/, 2017.
51. “Running TensorRT Optimized GoogLeNet on Jetson Nano,” [Online]. Available: https://jkjung-avt.github.io/tensorrt-googlenet/, May 2019.
52. “Deploying Customized Caffe Models on Intel Movidius,” [Online]. Available: https://movidius.github.io/blog/deploying-custom-caffe-models/, Jan. 2018.
53. R. Struharik, B. Vukobratović, A. Erdeljan and D. Rakanović, “CoNNA – Compressed CNN Hardware Accelerator,” *21st Euromicro Conference on Digital System Design (DSD)*, pp. 365-372, 2018.

54. C. Szegedy, V. Vincent, S. Ioffe, J. Shlens and Z. Wojna, "Rethinking the inception architecture for computer vision", *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 2818-2826, Jun. 2016.
55. K. He, X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition", *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 770-778, Jun. 2016.
56. 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:1602.07360*, 2016.
57. Song Han, Jeff Pool, John Tran, and William J. Dally, "C Learning both weights and connections for efficient neural networks," *In Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS'15)*, Vol. 1, pp. 1135–1143, 2015.
58. A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 4013-4021, Jun. 2016.
59. Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio, "Neural networks with few multiplications," *arXiv:1510.03009*, 2015.
60. C. Zhu, S. Han, H. Mao and W. J. Dally, "Trained ternary quantization," *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2017.
61. X. Zhang, Jianhua Zou, Xiang Ming, K. He and J. Sun, "Efficient and accurate approximations of nonlinear convolutional networks," *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1984-1992, 2015.
62. K. NGO, "FPGA hardware acceleration of inception style parameter reduced convolution neural networks," M.Sc. thesis, KTH royal institute of technology, sweden 2017.
63. Y.-H. Chen, T. Krishna, J. S. Emer and V. Sze, "Eyeriss: An energy–efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127-138, Jan. 2017.
64. S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, et al., "Eie: efficient inference engine on compressed deep neural network," *in Proceedings of the 43rd International Symposium on Computer Architecture*, IEEE Press, 2016.

Appendix A: GoogLeNet Layer Details

Table A.1: GoogLeNet layer details [17]

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Appendix B: ImageNet Dataset

Existence of huge datasets plays a vital role in the development of efficient computer vision algorithms using deep neural networks. In the early stages of artificial intelligence revolution, the availability of datasets have delayed the evolution of many algorithms till starting of internet revolution and Big Data.

ImageNet plays this role effectively which provided an open access dataset for the researchers and developers to develop more efficient applications. It's a huge database for over 14 million images. It has been originally created for computer vision research. However, it is used later in both industrial and research purposes. Moreover, it has been the first large scale image dataset over the world. Images are organized and labelled in main classes as shown in Figure B.1. Images are organized into 27 high-level categories with 21,841 subcategories. So that, ImageNet is a well-organized database that is used to benchmark machine learning models and algorithms.

The proposed hardware accelerator is designed to process GoogLeNet CNN. GoogLeNet CNN is firstly trained using ImageNet Dataset for 1000 standard classes. Both training and validation dataset of ImageNet have been firstly used. Finally, testing dataset is used to evaluate the model inference accuracy.

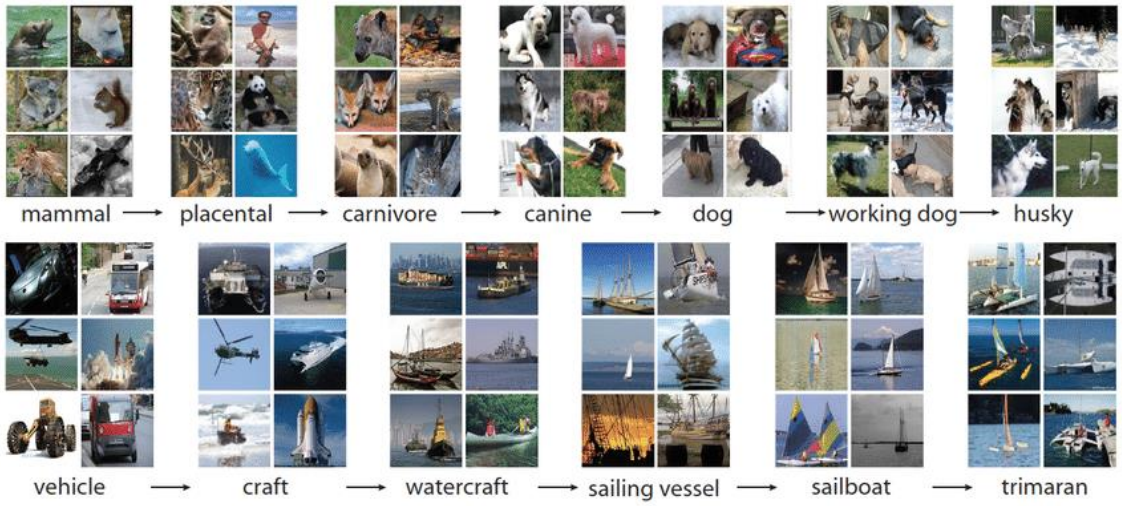


Figure B.1: A snapshot for ImageNet Dataset

Appendix C: Project Organization

This appendix shows the project organization for Verilog files on Vivado. It gives a closer view for the proposed hardware accelerator blocks arrangement.

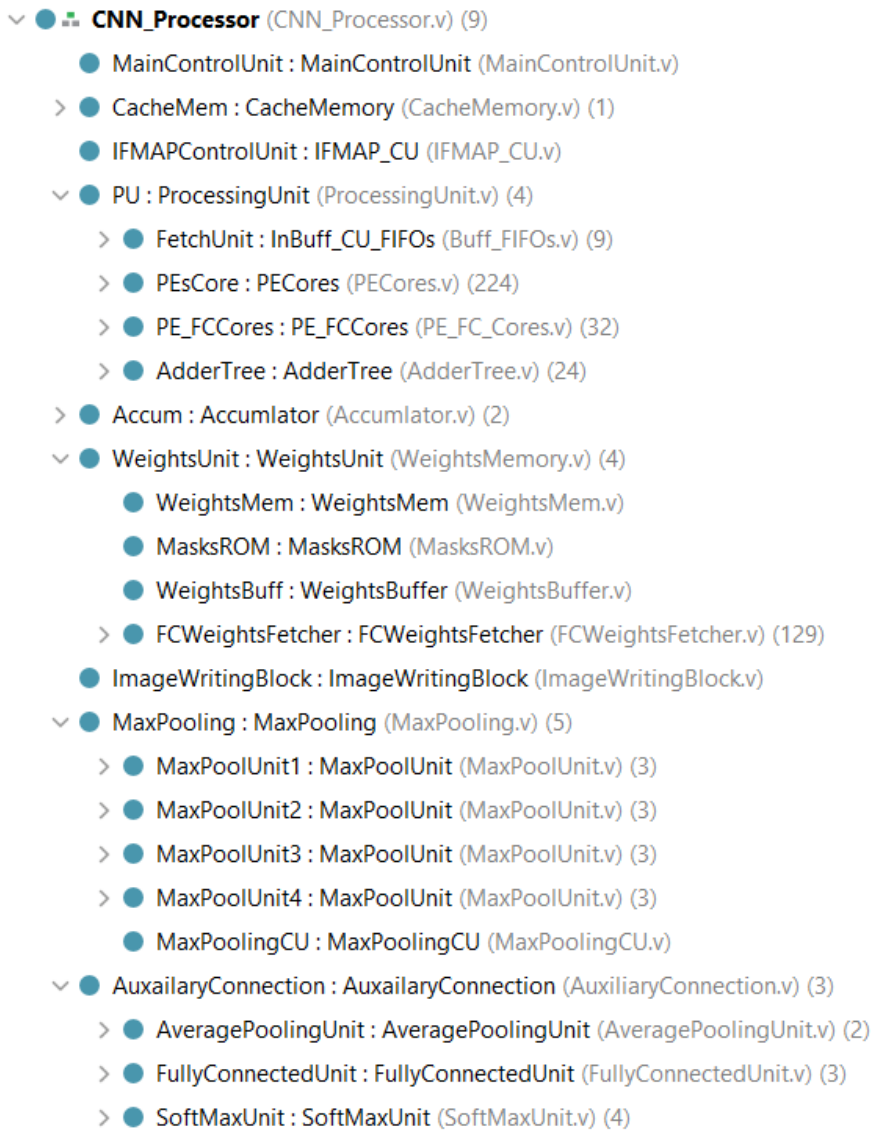


Figure C.1: A snapshot for top-level project organization in Vivado

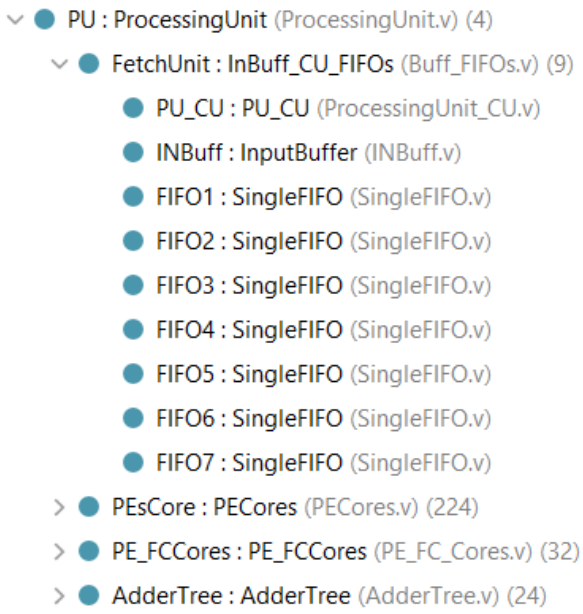


Figure C.2: A snapshot for processing unit organization in Vivado.

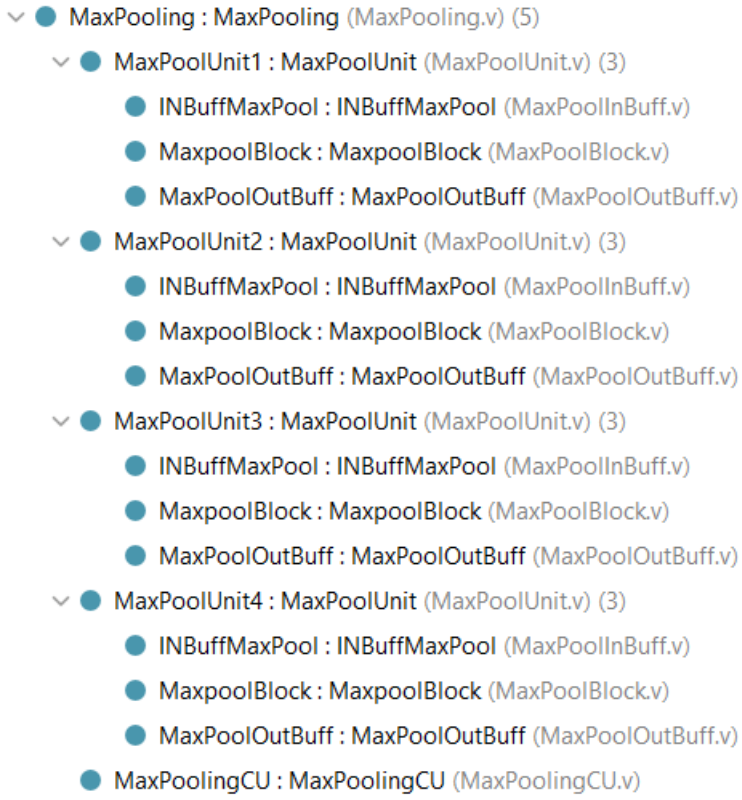


Figure C.3: A snapshot for maxpooling unit organization in Vivado

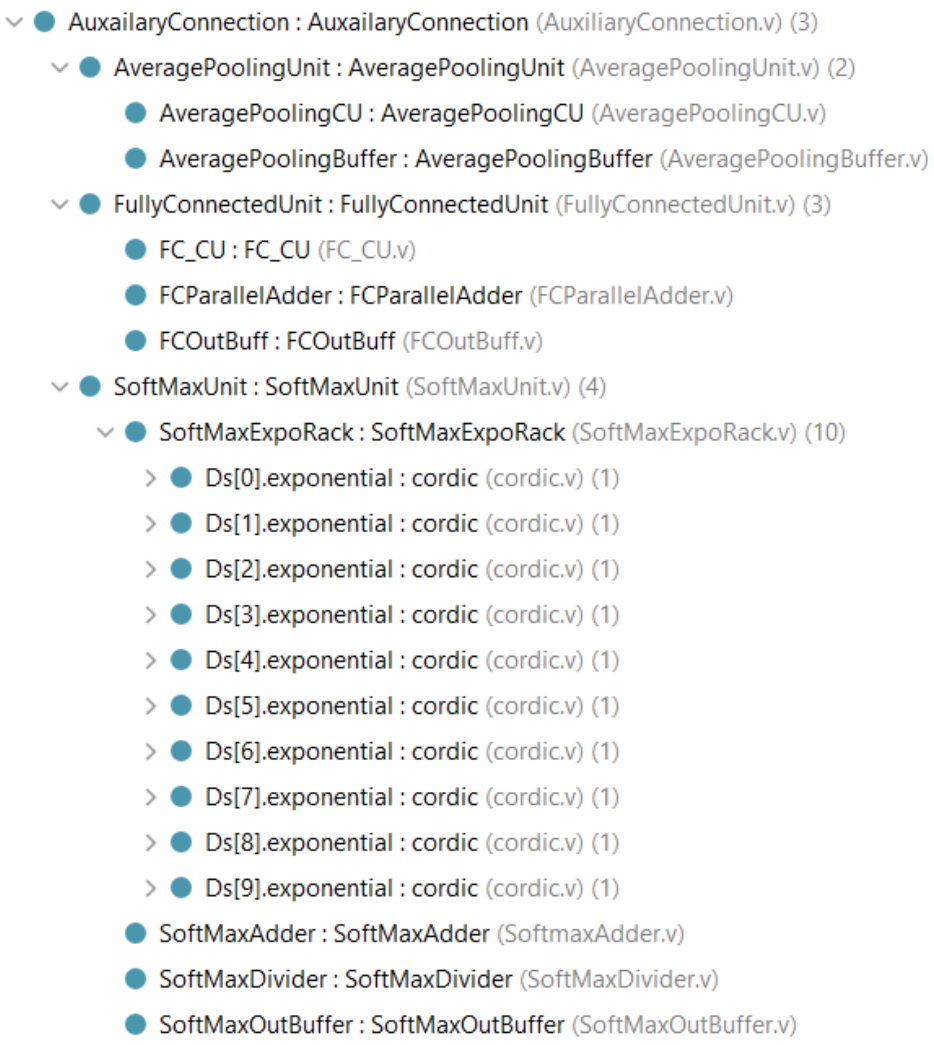


Figure C.4: A snapshot for auxiliary connection organization in Vivado

The following snapshot shows the first instantiations for parallel memory banks. The total number of memory banks is 256 as discussed in chapter 4.

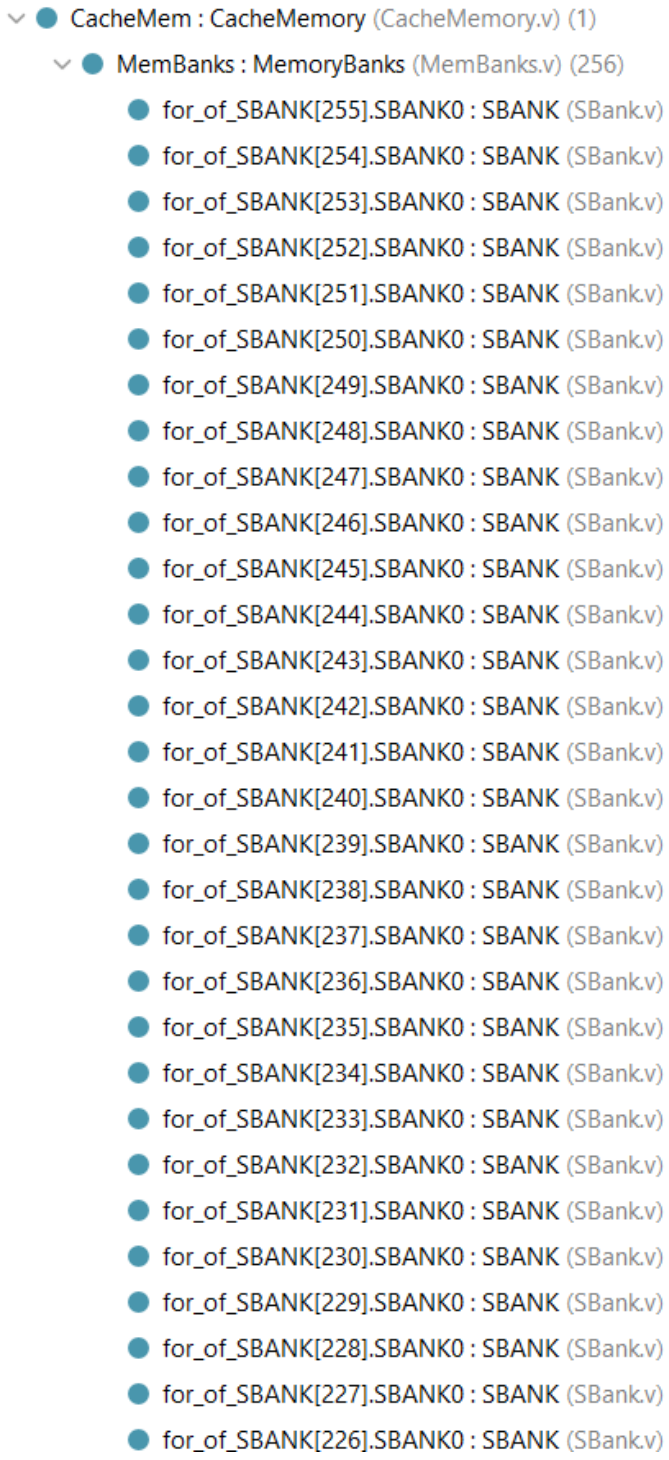


Figure C.5: A snapshot for the first part of memory Banks organization in Vivado.

The following snapshot shows the first instantiations for parallel PE cores. The total number of PE is 224 in addition to extra 32 PE cores for fully connected operation only as discussed in chapter 4.

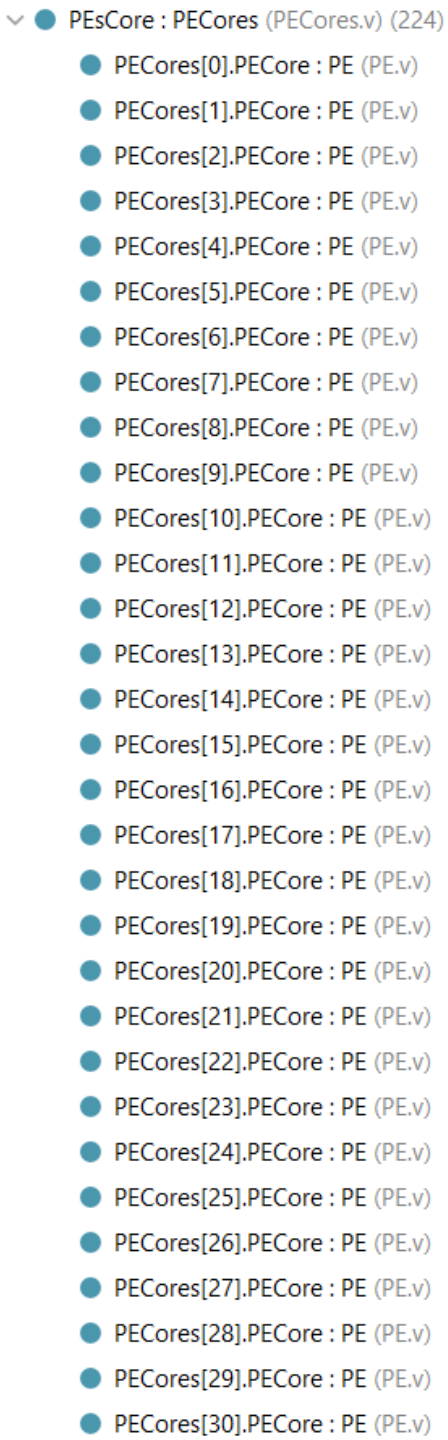


Figure C.6: A snapshot for the first part of PE Cores organization on Vivado.

Appendix D: Image Classification Example on The Proposed Accelerator

In this appendix an example for Image classification on the proposed hardware accelerator is presented, and the result is compared with the software result on python model. The experiment is made on an image for endian elephant as shown in Figure D.1. This image is chosen as it's from the hardest classification photos on both hardware and software model. The output prediction and class number for the proposed hardware accelerator are shown in Figure D.2 after mapping the output class number on Matlab. It's noted that the class prediction is 0.8505 on the hardware while its calculated 0.9015 by the software model. This is acceptable as long as the predicted class is correct "Indian Elephant" for the software and hardware models. The image is tested on the software model and the result is shown in Figure D.4. Also, the predicted class is shown in the classes list of ImageNet dataset as in Figure D.5.

Figure D.1: Input Image sample "Endian Elephant"

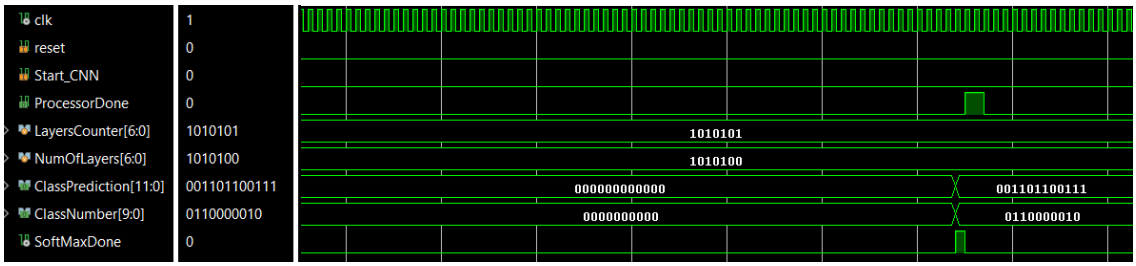


Figure D.2: Testing the image on The proposed Accelerator.

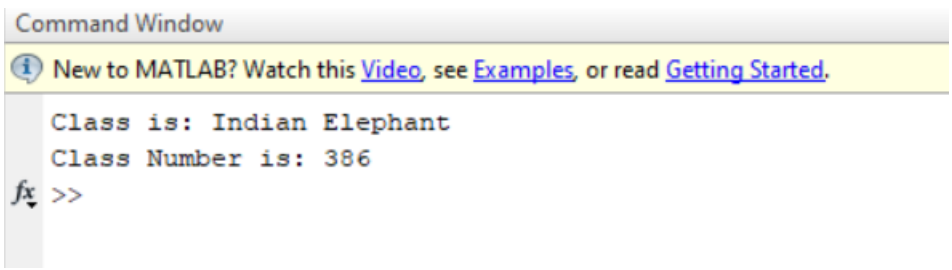


Figure D.3: Mapping class number of the hardware result to its name on Matlab.

```
Indian elephant 0.9015619158744812
tusker 0.058225732296705246
African elephant 0.021374277770519257
hippopotamus 0.012157320976257324
water buffalo 0.0023159412667155266

In [2]:
```

Figure D.4: Testing the same image on the software model (python model).

```
guenon
patas
baboon
macaque
langur
colobus
proboscis monkey
marmoset
capuchin
howler monkey
titi
spider monkey
squirrel monkey
Madagascar cat
indri
Indian elephant
African elephant
lesser panda
giant panda
barracouta
eel
coho
rock beauty
anemone fish
sturgeon
gar
lionfish
puffer
abacus
abaya
academic gown
accordion
acoustic guitar
aircraft carrier
airliner
```

Figure D.5: the Classes location in the ImageNet Dataset.

Appendix E: Virtex-7 FPGA

The Virtex-7 FPGA VC709 Connectivity Kit is a 40Gb/s platform for high-bandwidth and high-performance applications that includes all of the essential hardware, tools, and IP for efficient development. Figure E.1 shows Viretex-7 kit, while Table E.1 shows its available resources.

key features:

- Clocking:
  - Fixed Oscillator with differential 200MHz output used as the “system” clock for the FPGA.
  - Fixed Oscillator with differential 233.33MHz output used as the "memory" clock.
- Power
  - AC Power adapter (12V) or ATX
- Memory
  - DDR3 SODIMM (qty 2) - each with 4GB up to 933MHz / 1866Mbps
  - BPI Parallel NOR Flash: 32MB (256Mb)
- Control & I/O
  - User Push Buttons (x5)
  - User DIP Switch (8-position)
  - User LEDs (x8)
- **Configuration**
  - Onboard JTAG configuration circuitry to enable configuration over USB
  - BPI Parallel NOR Flash: 32MB (256Mb)
  -

Figure E.1: Viretex-7 FPGA kit

Table E.1: Virtex-7 FPGA Resources

Resource	Count
Logic Cells	693,120
Memory (Kb)	52,920
DSP Slices	3600
GTH 13.1Gb/s Transceivers	80
I/O Pins	1000

Useful Virtex-7 User Guides:

- https://www.xilinx.com/support/documentation/boards_and_kits/vc709/ug887-vc709-eval-board-v7-fpga.pdf
- https://www.xilinx.com/support/documentation/boards_and_kits/vc709/2014_3/ug966-v7-xt-connectivity-getting-started.pdf
- https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf

الملخص

اصبحت الخوارزميات CNNs الأكثر استخدامًا لتصنيف الصور. ومع ذلك، فإنه يتطلب عددًا كبيرًا من العمليات الحسابية وذاكرة تخزين ضخمة. هذا ينعكس بالطبع على استهلاك الطاقة وسرعة الأداء مما يتطلب تصميم معالجات بموصفات خاصة.

يقترح هذا العمل تصميمًا مسرعًا منخفض الطاقة للشبكات العصبية التلافيفية استنادًا على GoogLeNet CNN. تم تطبيق عدة تقنيات لضغط الذاكرة، مما أدى إلى تقليل حجم الذاكرة بمقدار ٥٧,٦ مرة مع خسارة دقة تصنيف بنسبة ٢,٦٪ فقط. تم استخدام FPGA BRAMs الداخلية لتخزين البيانات دون استخدام أي ذاكرة عشوائية خارجية. بالإضافة إلى ذلك، لا يستخدم هذا المسرع أيا من وحدات ال DSP لأنه استبدل جميع عمليات الضرب بعمليات تحويل بسيطة. بالإضافة الى ذلك، المسرع يتألف من ٢٢٤ عنصرًا متوازيًا وحقق دقة تصنيف بنسبة ٩١٪. استخدمت وحدات التحكم الموزعة التي جعلت من الممكن استخدام المسرع لتصنيف شبكات CNN أخرى.

اقترح هذا المسرع آلية جديدة لجلب البيانات، مما أدى إلى إعادة استخدام البيانات وتقليل استهلاك الطاقة. هذا بجانب استخدام عدة طرق للتقريب أو التحسين خلال تصميم كل وحدة من وحدات المسرع. لذلك، فإن المعالج يستهلك طاقة منخفضة مما جعله مناسبًا للعديد من التطبيقات. صنف المعالج ٢٥,١ إطارًا في الثانية لنموذج GoogLeNet باستخدام ٣,٩٢ واط مع تحسن في استهلاك الطاقة أكثر من التصميمات السابقة باستخدام الFPGA. قدم هذا المسرع تحسينًا في استخدام الطاقة بمعدل ٤٩,٥ مرة مقارنة بـ Intel Core-i7 و٧,٨ مرة مقارنة بـ NVidia GTX 1080Ti.

مهندس: أحمد جمال محمد عبدالمقصود

تاريخ الميلاد: ٢/ ٧ / ١٩٩٥

الجنسية: مصري

تاريخ التسجيل: ١ / ٣ / ٢٠١٩

تاريخ المنح: .... / .... / ٢٠٢١

القسم: هندسة الالكترونيات والاتصالات الكهربية

الدرجة: ماجستير العلوم

المشرفون:

أ.د احمد حسين محمد خليل

أ.م.د حسن مصطفى حسن مصطفى

الممتحنون:

أ.د. أحمد حسين محمد خليل	(المشرف الرئيسي)
أ.م.د. حسن مصطفى حسن مصطفى	(المشرف)
أ.د. محمد محمود رياض الغنيمي	(الممتحن الداخلي)
أ.د. أحمد حسن كامل مدين	(الممتحن الخارجي)

(هيئة الطاقة الذرية وجامعة النيل)

عنوان الرسالة:

تصميم موفر للطاقة لمسرع الشبكات العصبية التلافيفية عالي الأداء

الكلمات الدالة:

الشبكة العصبية التلافيفية، التعلم العميق، معالجة الصور، الذكاء الاصطناعي، مسرعات أداء

ملخص الرسالة:

أصبحت الخوارزميات CNNs الأكثر استخدامًا لتصنيف الصور. ومع ذلك، فإنها تتطلب عددًا كبيرًا من العمليات الحسابية وذاكرة تخزين ضخمة. علاوة على ذلك، أصبحت المعالجات ذات الاستخدام المتعدد لاتفي بمتطلبات استهلاك الطاقة وسرعة الأداء. لذلك يقترح هذا العمل تصميمًا مسرعًا منخفض الطاقة للشبكات العصبية التلافيفية استنادًا إلىGoogLeNet CNN. تم تطبيق عدة تقنيات لتقليل استهلاك الطاقة وحجم الذاكرة، حيث استخدمت وحدات FPGA BRAMs فقط بدون استخدام DRAMs خارجية. ايضاً، لا يستخدم هذا المسرع أي وحدات DSP مما يقلل من استهلاك الطاقة. حقق التصميم ٢٥,١ إطارًا في الثانية لتصنيف GoogLeNet باستهلاك ٣,٩٢واط فقط. يتكون التصميم من ٢٢٤ نواة متوازية ويحقق متوسط كفاءة حسابية بنسبة ٩١ ٪

تصميم موفر للطاقة لمسرع الشبكات العصبية التلافيفية عالي الأداء

اعداد

أحمد جمال محمد عبدالمقصود

رسالة مقدمة إلى كلية الهندسة – جامعة القاهرة

كجزء من متطلبات الحصول على درجة

ماجستير العلوم

في

هندسة الالكترونيات والاتصالات الكهربية

يعتمد من لجنة الممتحنين:

الاستاذ الدكتور: أحمد حسين محمد خليل (المشرف الرئيسى)

الاستاذ الدكتور: حسن مصطفى حسن مصطفى (مشرف)

الاستاذ الدكتور: محمد محمود رياض الغنيمي (الممتحن الداخلي)

الاستاذ الدكتور: أحمد حسن كامل مدين (الممتحن الخارجي)
(هيئة الطاقة الذرية وجامعة النيل)

كليــة الهندســة - جامعــة القاهـرة

الجيـزة - جمهوريـة مصـر العربيــة

٢٠٢١

تصميم موفر للطاقة لمسرع الشبكات العصبية التلافيفية عالي الأداء

اعداد

أحمد جمال محمد عبدالمقصود

رسالة مقدمة إلى كلية الهندسة – جامعة القاهرة
كجزء من متطلبات الحصول على درجة
ماجستير العلوم
في
هندسة الالكترونيات والاتصالات الكهربية

تحت اشراف

أ.د. أحمد حسين محمد خليل
هندسة الالكترونيات والاتصالات الكهربية
كلية الهندسة، جامعة القاهرة

أ.م.د حسن مصطفى حسن مصطفى
هندسة الالكترونيات والاتصالات الكهربية
كلية الهندسة، جامعة القاهرة

كليــة الهندســة - جامعــة القاهـرة
الجيـزة - جمهوريـة مصـر العربيــة
٢٠٢١

تصميم موفر للطاقة لمسرع الشبكات العصبية التلافيفية عالي الأداء

اعداد

أحمد جمال محمد عبدالمقصود

رسالة مقدمة إلى كلية الهندسة – جامعة القاهرة
كجزء من متطلبات الحصول على درجة
ماجستير العلوم
في
هندسة الالكترونيات والاتصالات الكهربية

كليــة الهندســة - جامعــة القاهـرة
الجيـزة - جمهوريـة مصـر العربيــة
٢٠٢١