ACCELERATED DEEP NEURAL NETWORKS USING FPGA
(ZynqNet Architecture)

A Graduation Project Report Submitted to
the Faculty of Engineering at Cairo University
in Partial Fulfillment of the Requirements for the
Degree of Bachelor of Science
in
Electronics and Electrical Communications Engineering

By

Amr Mohamed Gamal Eldin
Aya Hesham Omar
Gamal Saied Fadl
Mennat-Allah Ayman Ahmed
Omnia Essam Ahmed
Sara Mostafa Mohamed

Under supervision of

Dr. Hassan Mostafa

Faculty of Engineering, Cairo University

Giza, Egypt

August 2020

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ASIC	Application-Specific Integrated Circuit
BGR	Blue Green Red
BRAM	Block Random Access Memory
CAFFE	Convolutional Architecture for Fast Feature Embedding
CLB	Configurable Logic Block
CNN	Convolutional Neural Network
Concat	Concatenation
Conv	Convolution
CV	Computer Vision
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
DSD	Dense-Sparse-Dense training
DSP	Digital Signal Processing
e.g.	For Example
EIE	Efficient Inference Engine
FC	Fully Connected layer
FF	Flip-Flop
FIFO	First in First out
FPGA	Field Programmable Gate Arrays
HDL	Hardware Description Language
i.e.	In Other Words
I/O	Input/Output
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
LUT	Look Up Table
MAC	Multiply and Accumulate
MMCM	Mixed-Mode Clock Manager
PAR	Place and Route
RAM	Random Access Memory
ReLU	Rectified Linear Unit
ResNet	Residual Neural Network
RGB	Red-Green-Blue
ROM	Read Only Memory
SRL	Shift Register LUT
STA	Static Timing Analysis
TNS	Total Negative Slack
VGGNet	Visual Geometry Group network
VHDL	VHSIC(V.High Speed Integrated Circuit) Hardware Description Language
VIO	Virtual Input/Output
v.s.	Versus
WNS	Worst Negative Slack

Acknowledgments

First and Foremost praise is to Allah, the Almighty, the greatest of all, on whom ultimately we depend for sustenance and guidance. We would like to thank Almighty Allah for giving us opportunity, determination and strength to achieve this work. His continuous grace and mercy was with us throughout our life and even more during the tenure of this work.

Second, we want to thank our families for their support, tolerance and love during this year especially during the hard times they were always there having faith in what we do. We are grateful to our families, colleagues and friends for always motivating us, without them we wouldn't have come so far.

We want to thank our major advisor Dr. Hassan Mostafa for his caring about following up each stage in the project and his suggestions to solve some problems we faced during the project work.

Finally, we want to thank Mentor Graphics team represented in Dr. Eman Mandouh, Director of quality, Eng. Ziad Ibrahim, QA/test engineer at Questa static and formal and Eng. Sara Abd AlWahab, FPGA Prototyping QA/test Engineer, for providing their time, experience to help us overcome some obstacles we faced during some stages, and encouragement through the whole year.

Abstract

The convolutional neural network (CNN) is a class of deep learning neural networks. CNNs represent a huge breakthrough in image recognition. CNNS are most commonly used to analyze visual data and are frequently working in image classification application. The target in this field nowadays is to be used in real time applications such as Face Recognition - Search Engines, Recommender Systems, and Social Media.

The target in this project is to open the possibility of implementing any CNN architecture on FPGA, and focusing in accelerating the design and lowering power. ZynqNet is the chosen architecture due to its simplicity and its low number of parameters which is much less than other CNN architectures, which leads to the reduction in the number of resources needed to be implemented.

Target FPGA to be used is Virtex-7 available in VC709 board, it was chosen due to its high number of available resources which is suitable for CNN architectures.

Chapter 1: Introduction

1.1 Motivation

In recent years, the utility and effectiveness of artificial intelligence and deep learning have been proved in to solve many real-world computation-intensive problems. The motivation of these is to create an intelligent system that can automatically extract features and recognize a particular pattern. In addition to that it doesn't have to be trained on every single situation that could possibly exist, that makes deep-learning algorithms better suited in variable, situation-dependent decisions as in self-driving cars than traditional, rules-based approach. [1]

Image understanding is a very complex task for computers. But advanced Computer Vision (CV) systems can do image classification, object recognition and scene labeling. These CV systems become important in many applications in robotics, surveillance, smart factories and medical diagnostics. Significant progress has been made regarding the performance of these advanced CV systems. The availability of powerful computing platforms and the strong market pull have shaped a very fast-paced and dynamic field of research. But with development of artificial intelligence and deep learning, the previous approaches are replaced by machine learning concepts, where computers learn to understand images by looking at thousands of examples. These advanced learning algorithms, which are based on recent high-performance computing platforms as well as the abundance of training data available today are commonly referred to as deep learning. [2]

Convolutional Neural Network (CNN) is one of the brain-inspired algorithms that represent the most promising approach to image understanding and classification in CV systems, with significantly higher accuracy than traditional algorithms in many applications, such as image/video processing, face recognition, advances in medicine, machine language translation, autonomous driving and more. CNN consists of multiple layers of feature detectors and classifiers, which are adapted and optimized using techniques from machine learning. Neural Network (NN) is a computational model inspired by the operation of the brain, Artificial NNs use large amounts of simple elements which are organized in interconnected layers. Modern NNs have multiple layers; exceeding 100 these are called deep neural networks. The latest generations of high performance computing hardware have allowed the evaluation and training of CNNs deep to reach good performance in image understanding applications. State-of-the-art convolutional neural networks already rival the accuracy of humans when it comes to the classification of images. [3]

The main disadvantage of CNNs is the enormous computational complexity and to achieve accurate results, CNNs need many parameters (some over 100M parameters) and require huge amounts of computational resources and memory, they also offer significant potential for massive parallelization and extensive data reuse. The real time evaluation of a CNN may need billions or trillions of operations per second in order to provide image classification on a video stream. The most recent Graphics Processing Units (GPUs) can reach the level of performance that provides the needed effort for image segmentation and scene labeling. GPUs are expensive and power-hungry accelerators but efficiently process these networks, recently many applications such as embedded systems in self-driving cars need high energy efficiency and real-time performance. So there is a need to reduce the computational resources to reduce the used power and speed up the calculations.

Field-Programmable Gate Arrays (FPGAs) are among the most promising platforms that have been considered for efficient high-performance implementations of CNNs. FPGAs consist of versatile integrated circuits that provide hundreds of thousands of programmable logic blocks and a configurable interconnect, which enables the implementation of custom-made accelerator architectures in hardware. These have a lot of advantages with respect to embedded devices which are providing less computational power to CNNs, high energy efficiency, good performance, fast development round, and capability of reconfiguration.

The limitations of the computational resources and memory bandwidth of an FPGA platform must be considered. So, the accelerator structure must be carefully designed to make the computing throughput matches the memory bandwidth provided by the FPGA platform. It means that the performance is degraded due to the bottleneck of the memory bandwidth. As a result, it’s a must to find ways in order to reduce the number of the computations and the energy consumption. Acceleration approaches can be categorized into two main parts; General approaches (hardware independent) as reduction in precision, shared weights and data reuse. And customize the FPGA architecture to be suitable for the algorithm using pipelining, parallelism and increase the memory bandwidth.

1.2 Problem Statement

Deep Neural Networks (NN) have been a hot research topic in recent years. The key element of DNN is to explore the real time hardware implementation. However, it is required to specify where the NN is going to be implemented. Convolutional Neural Network (CNN) is the popular architecture of NN especially for image classification. CNN requires a huge number of computations to process a single image due to the convolution operation on the multiple dimensional arrays which represents a computational challenge for general purpose processors and consume a large amount of power. This required huge memory resources and consume a large amount of power. However, the high energy consumption is no big concern during the network's training phase - which typically takes place on a computer cluster - it poses a problem when the network needs to be evaluated on mobile hardware devices. Efficient implementation strategy of CNN is required to process more computations in real time.

The using of machine learning algorithms to extract and process the information had become popular in the recent years. It has been a race between GPU, ASIC and FPGA vendors to offer a HW platform that runs computationally intensive machine learning algorithms fast and efficiently, as the advanced machine learning applications is driven by Deep Learning, so Deep Learning is considered as the main comparison point.

The training of CNN models is commonly performed in floating-point representation on graphics processing units (GPUs) having thousands of cores and large external memory bandwidth. It does not require much effort to deploy existing models or train new ones on GPUs using various frameworks. Although GPUs performs batch computations which results in high performance, they are extremely power-hungry. This is affordable for training, which has no constraints on output latency and is carried out a limited number of times during the development phase. However, this is not ideal when it comes to inference for applications that have limited power budget and tight latency constraints such as mobile embedded platforms or self-driving car.

To achieve the best performance and energy-efficiency, many researchers have focused on building custom application-specific integrated circuits (ASICs) for accelerating CNNs inference workloads. Despite being an attractive solution, ASICs do not offer enough flexibility to accommodate the rapid evolution of CNN models and the emergence of new types of layers used in them. As well, the high non-recurring engineering (NRE) cost and time for design, verification and fabrication of a large ASIC chip makes it difficult to keep pace with the rapid model improvements in this space.

As a trade-off between performance, power-efficiency, and flexibility, FPGAs offer an interesting design point between GPUs and ASICs. FPGA-based accelerators provide high throughput, low power consumption, superior energy efficiency (Performance/Watt) compared to high-end GPUs, and configurability at a reasonable price, which provide low power consumption and recently have had much success in accelerating datacenter workloads in general and more specifically CNN inference tasks.

Processing speed is critical for many visual computing tasks. Many computer vision algorithms have high accurate results, but it’s too slow to produce results in real time. On the other hand, some algorithms with reduced accuracy is processed at camera frame rates, a more useful combination for real-time applications. Moreover, the capacity of hardware resources in the FPGA increases continuously and power consumption is reducing, making them more suitable for embedded applications, such as onboard vision and control for unmanned vehicles. CNNs offer state-of-the-art accuracy for many computer vision tasks. Their capabilities are generalizable to many different real-world applications. Real-world applications often require real-time responsiveness from the vision system. This Special Issue focuses on CNNs and their application to real-time computer vision tasks.

1.3 Solution Approach

As the main objective of this project is implementing CNN architecture on FPGA, so all the solution approaches will be directed into this field.

First for software approach, CNN inference needs huge amount of computations and memory, which limits the performance on embedded devices. In order to partially solve these problems, low precision fixed-point numbers are used to represent the CNN weights and activations. So fixed point representations were done reducing the number of bits from 64 bit floating into 16 bit taking into consecration that the accuracy wasn’t affected.

Second for Hardware approach, implementation of convolutional layers is to use MAC (Multiply and Accumulate) block, and by using parallelism, it was chosen that number of MACs in each layer depends on the number of output filters, also for optimization purposes, same layers in dimensions have resource sharing in MACs

For storage of weights and bias parameters of the architecture, weights of each layer are to be implemented in initialized 3D ROMs directly and it’s depth depends on the number of output filter and bias parameters are stored in initialized registers due to its low number, no need for external memory in this design and all parameters are stored on chip.

The approach in designing the storage of feature maps between the layers is using shared cache memory, the shared cache memory operates as input and output cache at the same time without need of external memory and one cycle of fetching is enough for the layer to start computing.

1.4 Organization

The following is a brief look on the contents of each chapter.

Chapter 2 provides background information on CNN, it discusses the main layers of the CNN with their operations, equations and functions and provides information about different CNN architectures, and also it discusses several methods to improve the efficiency of deep learning implementation. Finally, it discusses background on FPGAs, including a brief overview of FPGA internal components and design flow.

Chapter 3 provides information on the chosen CNN for the project which is ZynqNet and having a quick overview on its accuracy on ImageNet validation sets, the number of its layers and their arrangement and number of its parameters. It also shows the effect of changing the number of bits of the fixed point data propagating between layers on the accuracy and the chosen number of bits based on its effect on accuracy and compromising this effect with the number of resources utilized by the design.

Chapter 4 provides a discussion on the project’s chosen design with the details of how the convolution layers of ZynqNet are implemented, and introducing the different approaches of implementing kernel storage and intermediate storage, then the proper implementation of pooling layer, to end up with output prediction implementation for getting the index of the desired output.

Chapter 5 provides optimization techniques including optimization for ZynqNet implementation, timing and pipelining, power consumption, placement and routing optimizations. Also discusses the verification of the design by showing the testing strategy done to validate the functionality of the design.

Chapter 6 provides a discussion on the synthesis and which strategies give the best results; also constraints are handled including physical, timing and placement constraints, then implementation of the design is provided going through placing, routing then generating the bit stream.

Chapter 7 provides the results of implementation including utilization, timing analysis and power consumption, it also shows the burning of the bit stream on the chosen FPGA.

Chapter 8 concludes the previous work and introduces ideas for future work.

Chapter 2: Background and Related work

2.1 Convolutional Neural Networks

2.1.1 Neural Networks Overview

Deep Learning is a subfield of machine learning concerned with algorithms inspired by the structure of the brain which is called artificial neural networks.

The purpose of artificial neural networks is to achieve a very simplified model of the human brain. By having the artificial neural networks try to learn tasks mimicking the brain’s behavior. The brain consists of a large set of neurons which are specialized cell elements. These neurons are activated in response to the input, the activation of the neurons allows the network to detect and classify the patterns. Depending on certain input data, a neural network will try to calculate the probability that the data belong to a certain class (e.g., an object in a specific image). The neural network can be trained to recognize different classes by being provided a set of labeled training data which is called supervised learning.

2.1.2 Convolutional Neural Networks Overview

Convolutional Neural Networks (CNNs) are a special type of Neural Networks which are commonly used with visual data, which have shown state-of-the-art performance on various competitive benchmarks. The powerful learning ability of deep CNN is largely due to the use of multiple feature extraction stages (hidden layers) that can automatically learn representations from the data. The topology of CNN is divided into multiple learning stages composed of a combination of the convolutional layer, non-linear processing (ReLU) units, and subsampling (Pooling) layers. Each layer performs multiple transformations using a bank of convolutional kernels (filters). Convolution operation extracts locally correlated features by dividing the image into small slices (similar to the retina of the human eye), making it capable of learning suitable features. Output of the convolutional kernels is assigned to non-linear processing (ReLU) units, which not only helps in learning abstraction but also embeds non-linearity in the feature space. This non-linearity generates different patterns of activations for different responses and thus facilitates in learning of semantic differences in images. Output of the non-linear function (ReLU) is usually followed by subsampling (Pooling), which helps in summarizing the results and also makes the input invariant to geometrical distortions. [4]

2.1.3 Convolutional Neural Networks Layers

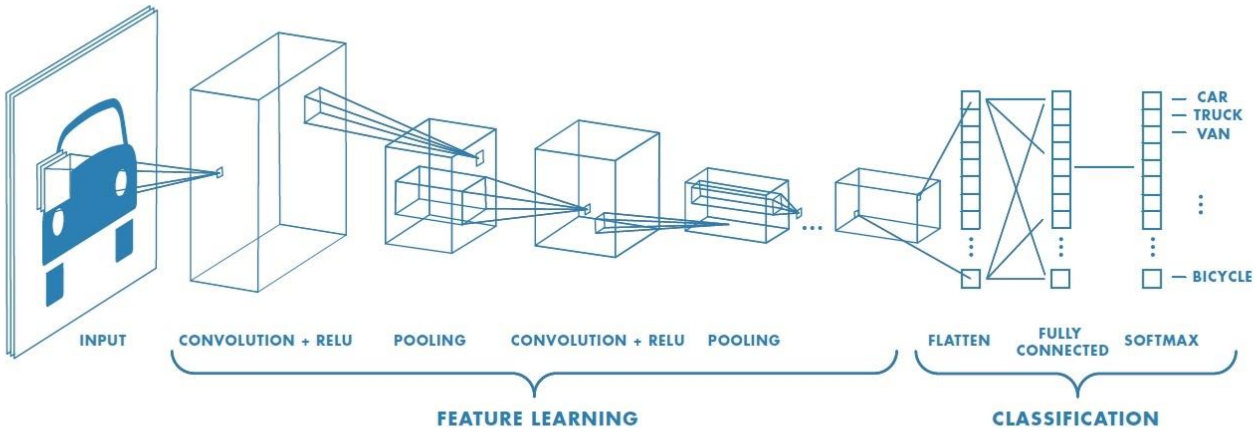


Figure 2-1 Convolutional Neural Networks Layers

2.1.3.1 Convolutional Layer

As in Figure 2-1, the first layer in a CNN is a Convolutional Layer which consists of set of trainable filters. The convolutional layer takes in square patches of pixels and passes them through a filter; the filter (or kernel) is used to detect patterns in the pixels. The convolutional layer receives N feature maps as input; each input feature map is convolved by a shifting window with k x k kernel (filter) to generate one element in one output feature map. If the input RGB image has dimensions n_H x n_W x 3 and the filter has dimensions k x k x 3, then the convoluted output is of dimensions (n_H-k+1) x (n_W-k+1). The stride of the shifting window is S, which is normally smaller than K. A total of M output feature maps will form the set of input feature maps for the next convolutional layer as shown in Figure 2-2, convolutional layer detect low-level features of images i.e.: edges and colors, and by stacking a number of convolutional layers, the network hierarchically learns high-level features of the image. [5]

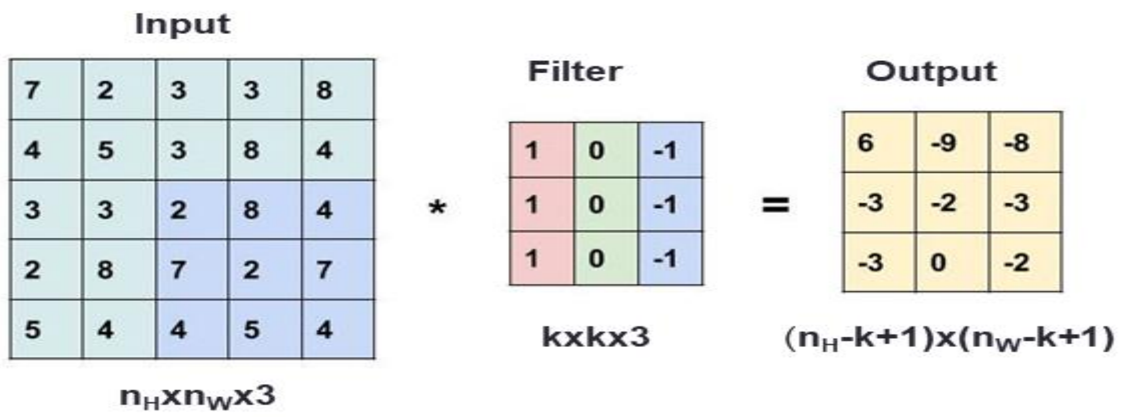
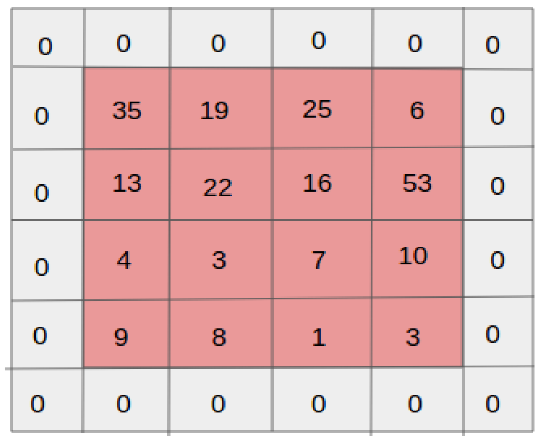


Figure 2-2 Convolution Operation

2.1.3.2 Padding

Padding is usually used to preserve the information that exists near the edge of the image; also it's used to preserve the original input size. No padding is called Valid Padding, and using padding to preserve the original input size is called Same Padding. The convoluted output is now of dimensions (n_H-k+2p+1) x (n_W-k+2p+1) as in Figure 2-3.



0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

Figure 2-3 Padding Operation

2.1.3.3 Stride

Stride is the size of the step in which the filter moves with across the image until it reaches the upper right-hand corner .Stride is used mainly to decrease the output size so processing will be easier. The convoluted output is now of dimensions $\left(\left\lfloor\frac{nH-k+2p}{s}\right\rfloor+1\right) x \left(\left\lfloor\frac{nw-k+2p}{s}\right\rfloor+1\right)$ as the example for striding by 2 is presented in Figure 2-4.

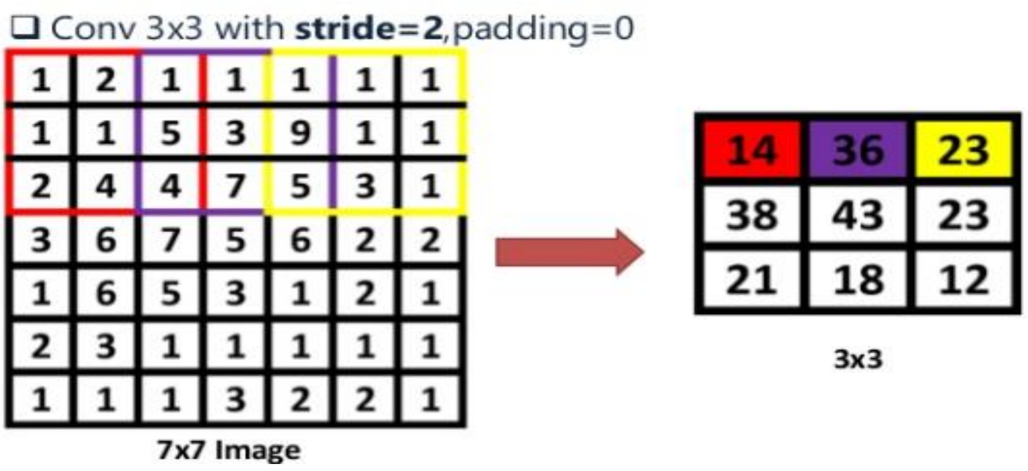


Figure 2-4 Striding Operation

2.1.3.4 ReLU

The ReLU layer shown in Figure 2-5 introduces a non-linear operation. The ReLU performs after every Convolutional layer. Its output is given by; max (0, input), the purpose of ReLU is to introduce nonlinearity in the CNN after linear operation of convolution, since most of the real-world data the network required to learn is nonlinear to generalize or adapt with variety of data.

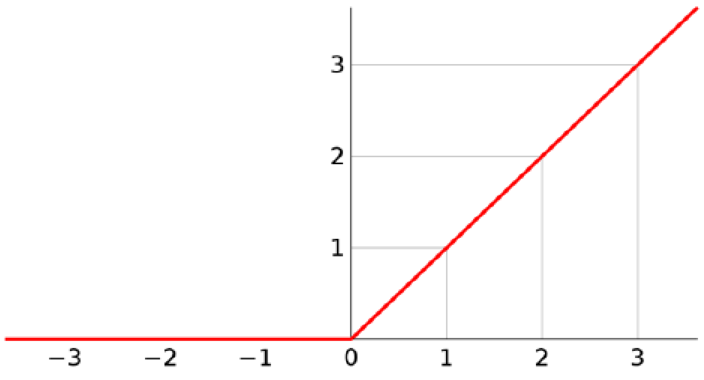


Figure 2-5 ReLU Function

2.1.3.5 Pooling Layer

As in Figure 2-6, the Pooling layer (also called sub-sampling) reduce the dimensionality of each feature map of its input feature maps, but retain the most important information. The number of output feature maps is identical to that of input feature maps, while the dimensions of each feature map scale down according to the size of the sub-sampling window (called also kernel). For example, for a pooling layer there is average or maximum. Max-pooling being the most popular, this basically takes a filter $P \ x \ P$ and a stride of length S, it then applies it to the input volume and outputs the maximum number in every sub-region that the filter convolves around. [6]

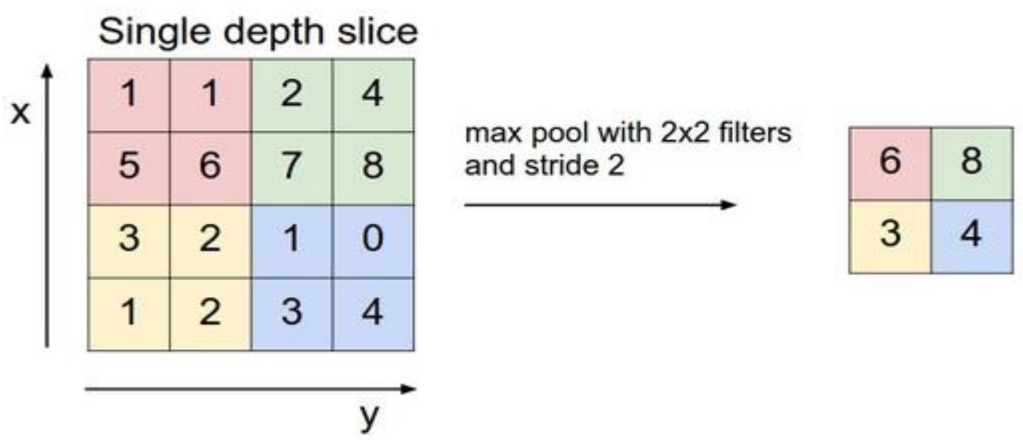


Figure 2-6 Maximum Pooling Operation

2.1.3.6 Fully Connected Layer (FC)

The way this fully connected neural network layer (FC) works is that it looks at the output of the previous layer (which represent the activation maps of high level features) and determines which features most correlate to a particular class by unrolling the input features and the weights and multiply them and outputs an N dimensional vector where N is the number of classes .Also this layer is followed by soft max to show the most correlated class to the input as shown in Figure 2-7 which is an example for image classification. [6]

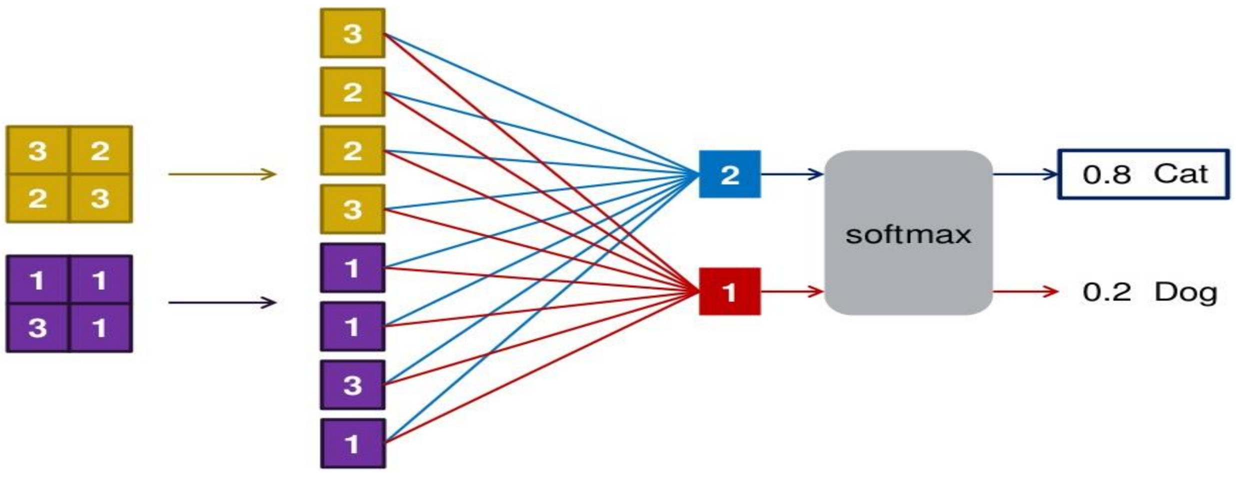


Figure 2-7 Fully Connected Operation

2.1.4 Convolutional Neural Networks Architectures

Various architectures were designed to efficiently use the layers of CNN in order to achieve successful implementation of specific application such as image classification, CNN models can be categorized to Classical architectures which were comprised simply of stacked layers, and Modern architectures which concentrated on innovative ways for efficient learning.

2.1.4.1 Classic network architectures:

2.1.4.1.1 LeNet-5:

LeNet-5 by LeCun et al. in 1998 is considered a pioneering CNN with 7 layers and was used by several banks to classify hand-written digits written on cheques, input is 32 x 32 greyscale images and to use higher resolution images, it will require more convolutional layers which aren't available in LeNet. [7]

2.1.4.1.2 AlexNet:

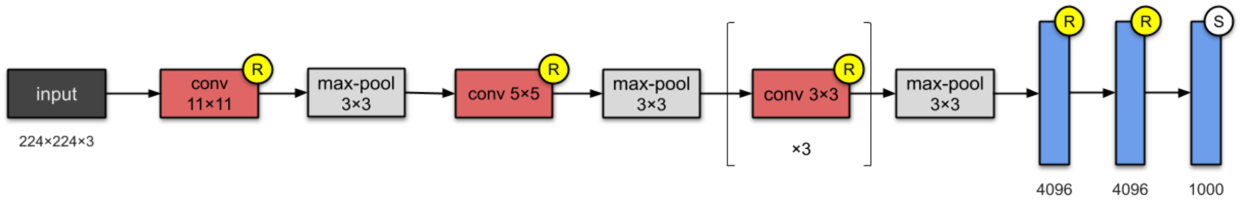


Figure 2-8 AlexNet Model

AlexNet by Alex Krizhevsky et al. is considered the reason why CNN is used in computer vision applications as it proved its capability to perform efficiently in that domain, AlexNet won the 2012 ILSVRC competition by a large margin (19.73% VS 26.2% (second place) top-5 error rates). [8]

As in Figure 2-8, It starts with 227 x 227 x 3 images and the next convolution layer applies 96 of 11 x 11 filter with stride of 4, Next layer is a pooling layer which applies max pool by 3 x 3 filter along with stride 2. It goes on and finally reaches FC layer with 9216 parameter and the next two FC layers with 4096 node each. At the end, it uses Softmax function with 1000 output classes. It has 60 million parameters so it has considerably large number of parameters.

AlexNet success was because of new methods used that was not adopted at that time such as:

1-Using ReLU as the nonlinearly after the convolution layers instead of Sigmoid and tanh functions that were commonly used which increased the speed greatly.

2-Using maximum pooling instead of traditionally used average pooling.

3-Using dropout method between fully connected layers in order to improve the generalization error instead of using ordinary regularization.

2.1.4.1.3 VGGNet:

VGGNet which stands for Visual Geometry Group was developed by Simonyan and Zisserman and achieved second place in 2014 ILSVRC competition, considered a simpler form of a deep CNN because of its uniform architecture and simplicity, most popular VGGNet architectures are VGG-16 as in Figure 2-9 and VGG-19. [7]

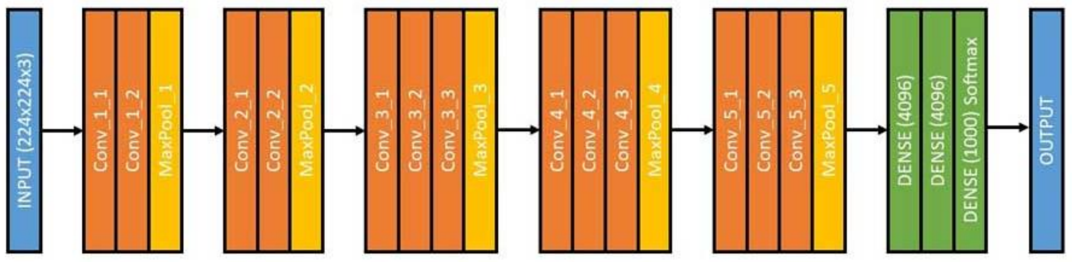


Figure 2-9 VGGNet Model

VGGNet has three simple rules of thumbs to be followed:

1-Each convolutional layer has the same parameters which are; kernel size of 3 x 3, stride of 1 and same padding to preserve the input size, number of filters is the difference between convolutional layers.

2-Convolutional layers are stacked with increasing number of filters, i.e., if layer-1 has 16 filters, then layer-2 must have 16 or more filters.

3-Each maximum pooling layer has the same parameters which are; window size of 3 x 3 and stride of 2, so the image size is halved after each pooling layer.

VGG-16 reached a top-5 error of 11.32%. However, the network contains almost 140 million parameters and one forward pass requires nearly 16 billion MACC operations

2.1.4.2 Modern network architectures:

2.1.4.2.1 GoogleNet:

GoogleNet shown in Figure 2-10 was developed by Christian Szegedy et al. from Google and was winner of the 2014 ILSVRC competition using Inception modules with smaller convolutions to decrease computations and number of parameters to 4 million only, it achieved top-5 error rate of 6.67%. [9]

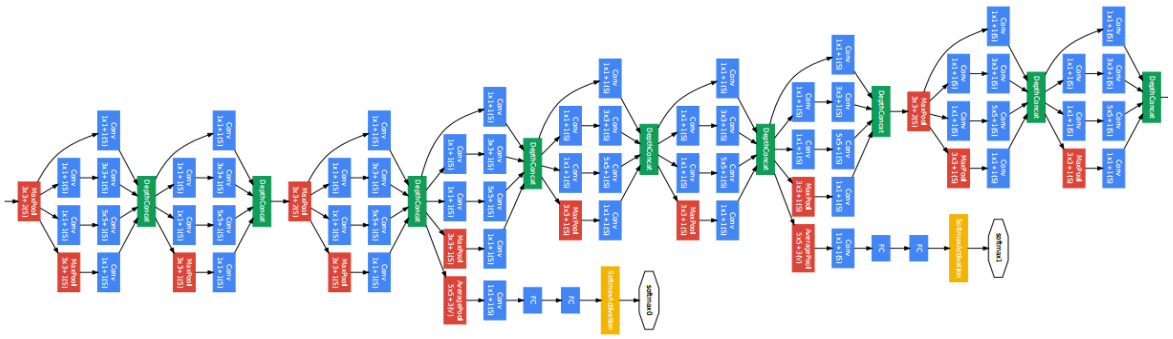


Figure 2-10 GoogleNet Model

2.1.4.2.2 ResNet:

Residual Neural Network (ResNet) in Figure 2-11 was introduced by by Kaiming He et al from Microsoft Research, skip connections were used in which helped in making CNN much deeper, there are multiple versions of ResNet architectures but most commonly used is ResNet152 which won 2015 ILSVRC competition and consists of 152 layers, it achieves top-5 error rate of 3.57% and contains only 60 million parameter which is considered small number its huge number of layers. [9]

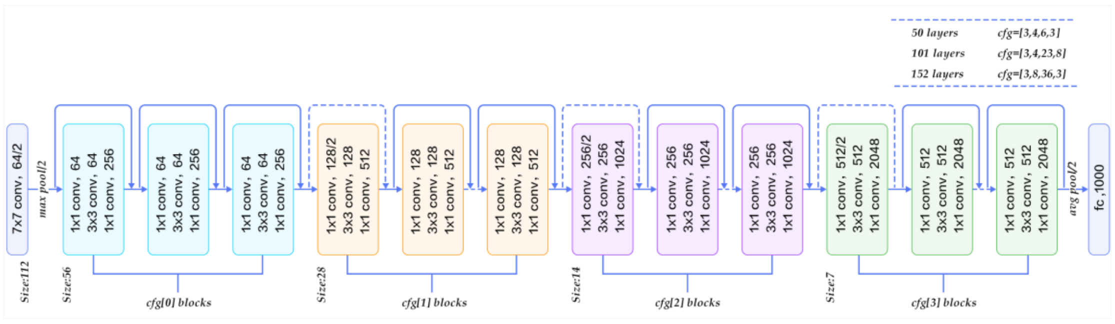


Figure 2-11 ResNet Model

2.1.4.2.3 SqueezeNet:

SqueezeNet by Forrest Iandola et al. concentrated mainly on reduction of the number of parameters used instead of concentrating in increasing accuracy so it reached accuracy similar to AlexNet but with 50x less parameters, SqueezeNet doesn’t contain fully connected layers so it begins with a standalone convolution layer (conv1), followed by 8 Fire modules (fire2–9) which will be explained briefly in chapter 3, ending with a final conv layer (conv10), global average pooling and Softmax output unit which achieved top-5 error rate of 17.3% with only 1.24 million parameters, as in Figure 2-12, SqueezeNet have other versions with skip connections and complex skip connections that were used to decrease error rate more. [10]

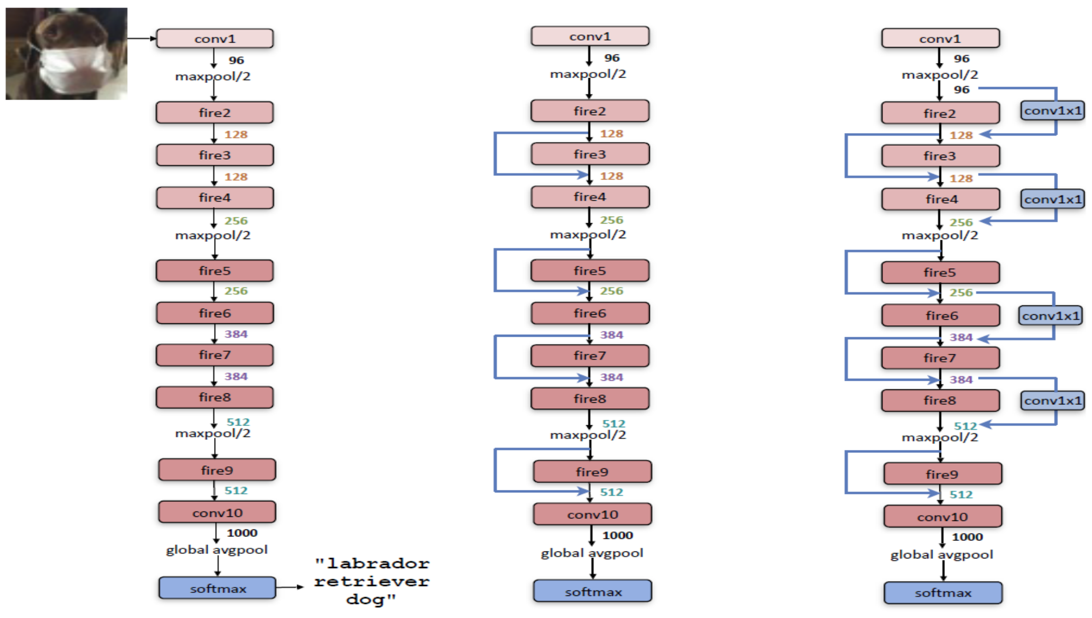
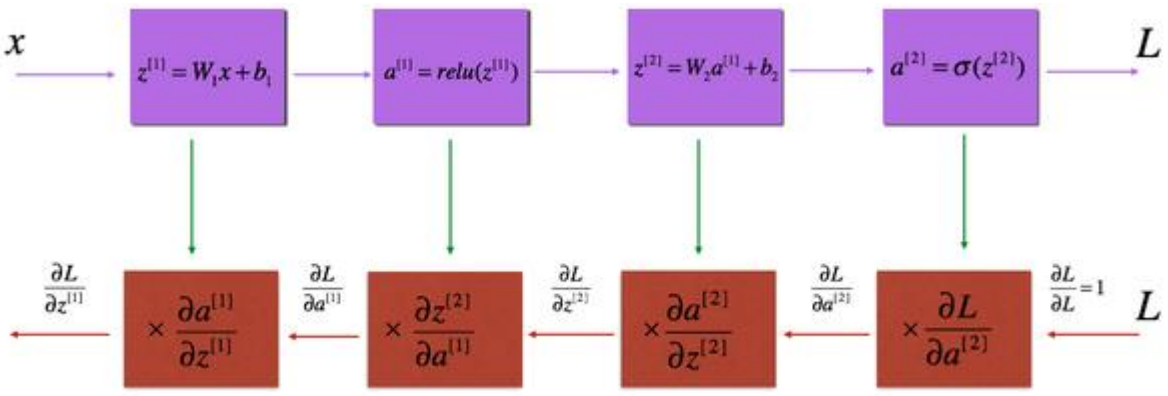


Figure 2-12 SqueezeNet Model

2.2 Training Process

The basic learning that has to be done in neural networks is training neurons when to get activated. Each neuron should activate only for particular type of inputs and not all inputs. Therefore, by propagating forward, it is noticed how well the neural network is behaving and find the error. After finding out that the network has error, backpropagation is applied and a form of gradient descent is used to update new values of weights. Then, forward propagation is applied again to see how well those weights are performing and then the weights are updated using backpropagation. This will go on until reaching some minima for error value.



x
$z^{[1]} = W_1 x + b_1$
$a^{[1]} = relu(z^{[1]})$
$z^{[2]} = W_2 a^{[1]} + b_2$
$a^{[2]} = \sigma(z^{[2]})$
L
$\frac{\partial L}{\partial z^{[1]}}$
$\times \frac{\partial a^{[1]}}{\partial z^{[1]}}$
$\frac{\partial L}{\partial a^{[1]}}$
$\times \frac{\partial z^{[2]}}{\partial a^{[1]}}$
$\frac{\partial L}{\partial z^{[2]}}$
$\times \frac{\partial a^{[2]}}{\partial z^{[2]}}$
$\frac{\partial L}{\partial a^{[2]}}$
$\times \frac{\partial L}{\partial a^{[2]}}$
$\frac{\partial L}{\partial L} = 1$
L

Figure 2-13 Forward and Backward Propagation

2.2.1 Forward Propagation

In forward Propagation as in Figure 2-13, in 1st row, input X is provided to each neuron and two functions are calculated, one is linear multiplication i.e. $Z = W \times X + b$ and the other is activation function a = ReLU(z), different activation functions can be used, then it will forward through every layer and predicted output is obtained.

2.2.2 Backward Propagation

Back propagation is a technique to reduce the loss i.e. (Actual o/p-predicted o/p) by updating the parameters weight, bias by using an algorithm called Gradient descent. For example in Figure 2-13 in 2nd row last column, Loss(L) is partially differentiated w.r.t a[2] but a[2] depends on z[2], again z[2] depends on weight w[1], activation a[1], and bias b[1], so gradients of a[1], w[1] and b[1] is calculated w.r.t

Loss(L). Then by using Gradient Descent algorithm, weight and bias in that layer are updated but again a[1] depends on calculation of z[1]. Above procedure will be repeated till first layer. Finally, it will propagate backside to reduce the loss by calculating all parameters gradients w.r.t Loss(L), and update them by using Gradient Descent algorithm.

2.2.3 Loss Function

A loss function can be defined in many different ways but a common one is MSE (Mean Squared Error), which are half times (actual - predicted) squared.

$$E_{total} = \Sigma \frac{1}{2}(target - output)^2$$

The predicted label (output of the CNN) must be the same as the training label (This means that the network got its prediction right). In order to achieve this, it's a must to minimize the amount of loss (error). It just an optimization problem in calculus to find out which inputs (weights) most directly contributed to the loss (or error) of the network as shown in Figure 2-14.

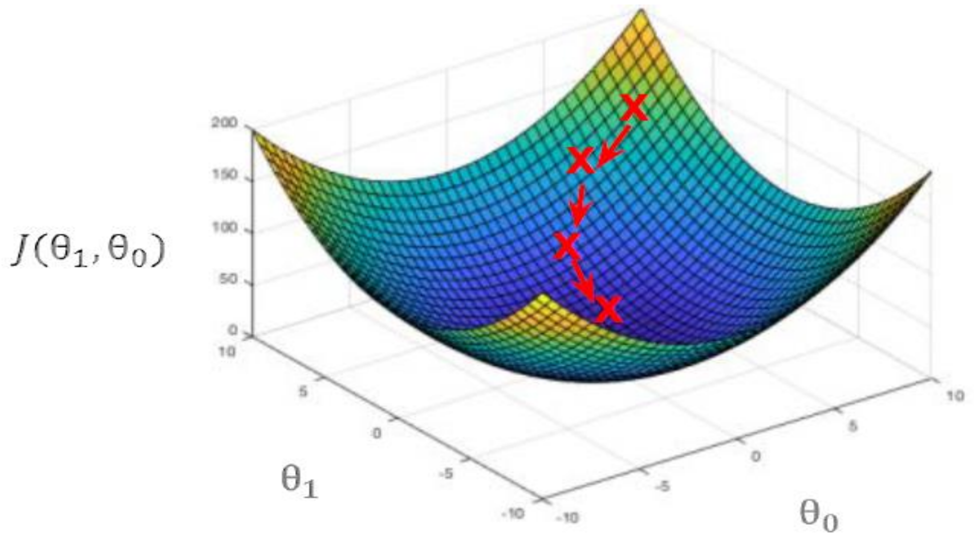


Figure 2-14 Loss Function

2.3 Methods to improve the efficiency of deep learning implementation

2.3.1 Pruning

Pruning is defined as discarding less important neuron without changing the original network structure as shown in Figure 2-15, to make the network size smaller and to alleviate over-fitting, without affecting the accuracy of original network.

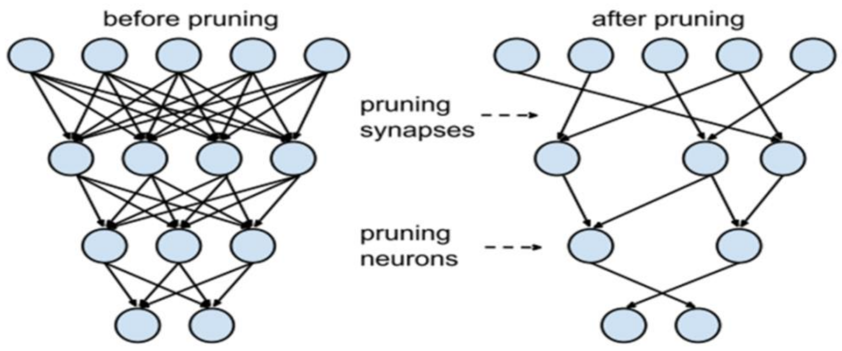


Figure 2-15 Synapses and neurons before and after pruning

Pruning method has three steps as shown in Figure 2-16; the first step is learning the connectivity via normal network training to learn which connections are important. Unlike conventional training that used to learn the final values of the weights. The second step is to prune all connections with weights below a threshold are removed from the. The final step is retraining the network to learn the final weights for the remaining sparse connections. The final step important is preserving the accuracy as shown in Figure 2-17. [11]

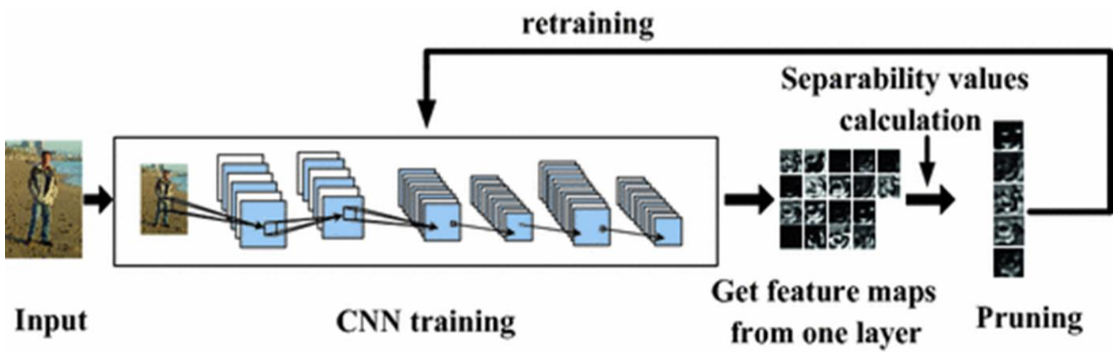

retraining
Separability values calculation
Input
CNN training
Get feature maps from one layer
Pruning

Figure 2-16 Pruning process

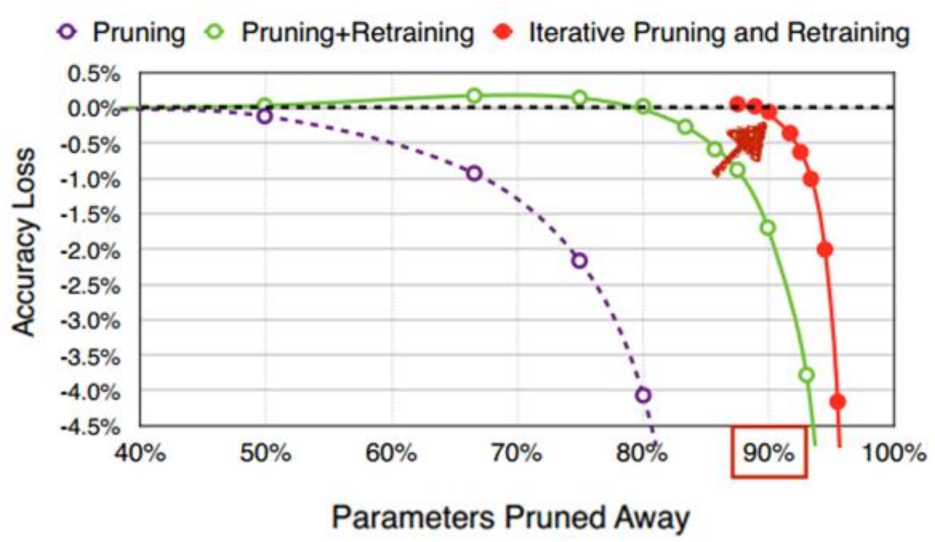


Figure 2-17 Accuracy loss Vs. Parameters pruned away

The difference between dropout and pruning is that in dropout, each parameter is probabilistically dropped during training, but will come back during inference. In pruning, parameters are dropped forever after pruning and have no chance to come back during both training and inference.

After pruning, the storage requirements of AlexNet and VGGNet are small enough that all weights can be stored on chip, instead of off-chip DRAM which takes orders of magnitude more energy to access as shown in Table 2-1. [12]

Table 2-1 Network's parameters and accuracy 1 before and after pruning

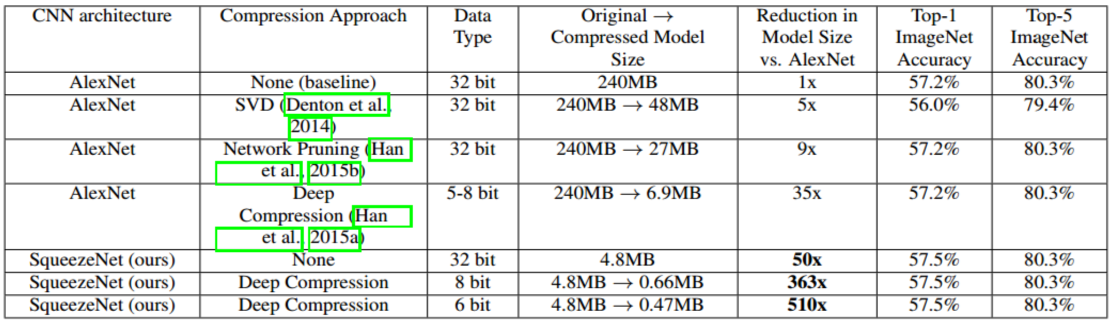
Network	Top-1 Error	Top-5 Error	Parameters	Compression Rate
LeNet-300-100 Ref	1.64%	-	267K	
LeNet-300-100 Pruned	1.59%	-	**22K**	**12**×
LeNet-5 Ref	0.80%	-	431K	
LeNet-5 Pruned	0.77%	-	**36K**	**12**×
AlexNet Ref	42.78%	19.73%	61M	
AlexNet Pruned	42.77%	19.67%	**6.7M**	**9**×
VGG-16 Ref	31.50%	11.32%	138M	
VGG-16 Pruned	31.34%	10.88%	**10.3M**	**13**×

2.3.2 Quantization

Network quantization and weight sharing compresses the pruned network by reducing the number of bits required to represent each weight. In the context of deep learning, the predominant numerical format used for research and for deployment has so far been 32-bit floating point. However, the desire for reduced bandwidth of deep learning models has driven research into using lower-precision numerical formats. It has been extensively demonstrated that weights and activations can be represented using 8-bits without getting significant loss in accuracy. [12]

Applying deep compression with 8-bits quantization on SqueezeNet yields a 0.66MB model with equivalent accuracy to AlexNet. And applying deep compression with 8-bits quantization and 33% sparsity on SqueezeNet yields a 0.47MB model with equivalent accuracy as shown in Table 2-2. [10]

Table 2-2 Comparing SqueezeNet to model compression approaches



CNN architecture	Compression Approach	Data Type	Original → Compressed Model Size	Reduction in Model Size vs. AlexNet	Top-1 ImageNet Accuracy	Top-5 ImageNet Accuracy
AlexNet	None (baseline)	32 bit	240MB	1x	57.2%	80.3%
AlexNet	SVD (Denton et al., 2014)	32 bit	240MB → 48MB	5x	56.0%	79.4%
AlexNet	Network Pruning (Han et al., 2015b)	32 bit	240MB → 27MB	9x	57.2%	80.3%
AlexNet	Deep Compression (Han et al., 2015a)	5-8 bit	240MB → 6.9MB	35x	57.2%	80.3%
SqueezeNet (ours)	None	32 bit	4.8MB	**50x**	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	8 bit	4.8MB → 0.66MB	**363x**	57.5%	80.3%
SqueezeNet (ours)	Deep Compression	6 bit	4.8MB → 0.47MB	**510x**	57.5%	80.3%

2.3.3 Low Rank Approximation

Low rank approximation for convolution layer means that a convolution layer with d filters with filter size (k x k x c) is decomposed to two layers one with d’ filters with filter size (k x k x c) and another with d filters with filter size (1 x 1 x d’) as shown in Figure 2-18. So that it’s like breaking a complicated problem into two separate small problems.

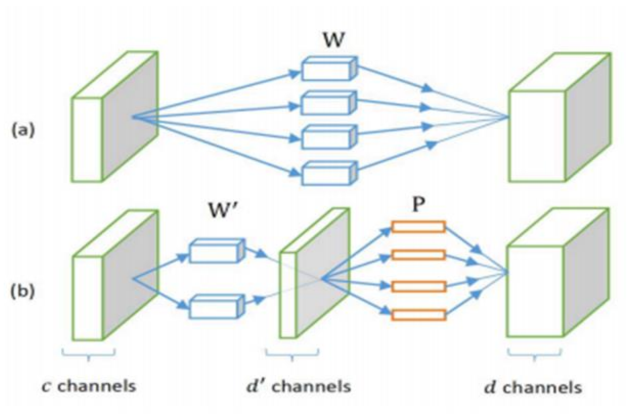


Figure 2-18 Low rank approximation for conv layer

There is also low rank approximation for fully connected layer but its use is less than that used for convolution because the convolution is more complex as shown in Figure 2-19.

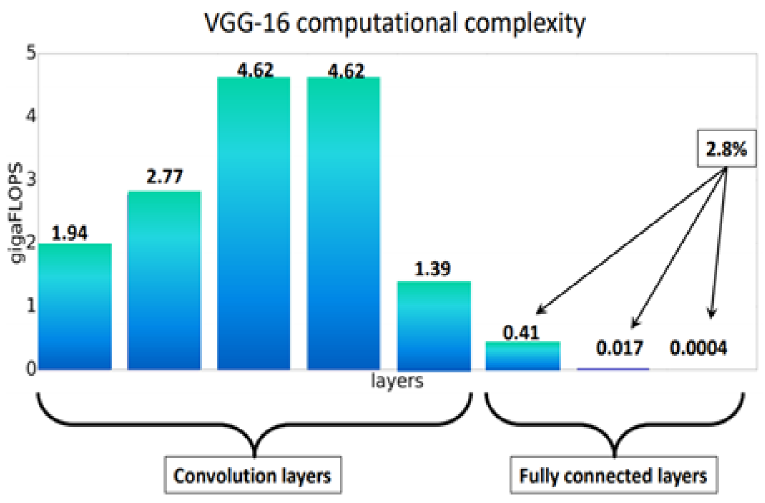


Figure 2-19 Computational complexity graph for VGG-16

2.3.4 Late down sampling

Each convolution layer in a convolutional network produces an output activation map with a spatial resolution that is at least 1x1 and often much larger than 1x1. The height and width of these activation maps are controlled by: (1) the input data size (2) down sample layers in the architecture. Down sampling is often engineered into CNN architectures by pooling layers as shown in Figure 2-20 or by setting the (stride > 1) in some of the convolution. [10]

If most layers in the network have a stride of 1, and the strides greater than 1 are moved towards the end of the network, then large activation maps will be in many layers in the network. That will cause higher classification accuracy. [10]

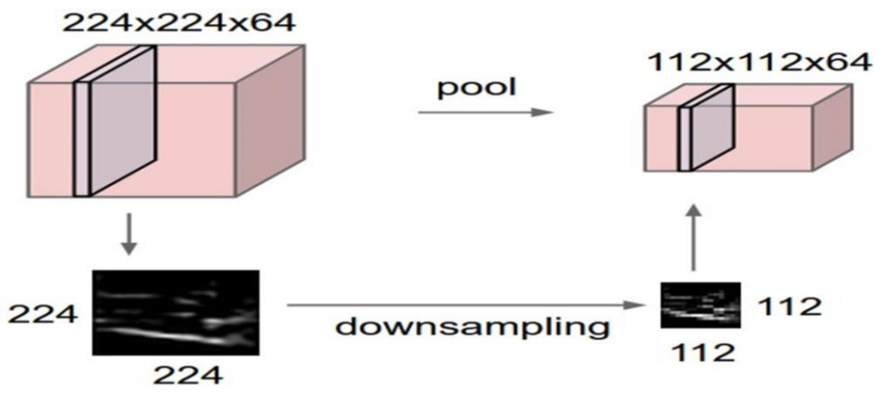


Figure 2-20 Pooling layer

2.3.5 DSD: Dense-Sparse-Dense Training

DSD produces same model architecture but can find better optimization performance by regularizing deep neural networks. The first step “Dense” is training a dense network to learn important weights. The second step “Sparse” is pruning the network and retraining the network to learn the final weights for the remaining sparse connections. The final step “re-Dense” is increasing the model capacity by re-initializing the pruned parameters from zero and retrain the whole dense network as shown in Figure 2-21. [13]

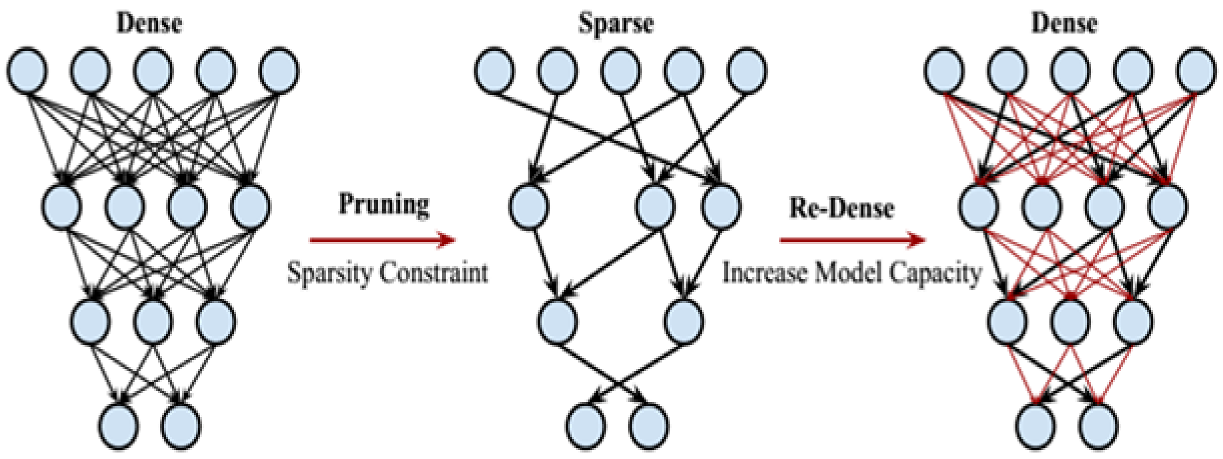


Figure 2-21 Dense-Sparse-Dense Training Flow

DSD and Dropout both are used to prevent over-fitting and regularize the network. But the difference is that DSD training learns with a deterministic data driven sparsity pattern but Dropout uses a random sparsity pattern at each SGD iteration.

2.3.6 Sparsing

The result of multiplying any number by zero is zero so computing it isn’t necessary and can be discarded. That will save memory by reducing computations. This method is used in EIE model (the First DNN Accelerator for Sparse, Compressed Model), the reduction of computations and memory as a result of sparsing weights and activations is shown in Figure 2-22. [14]

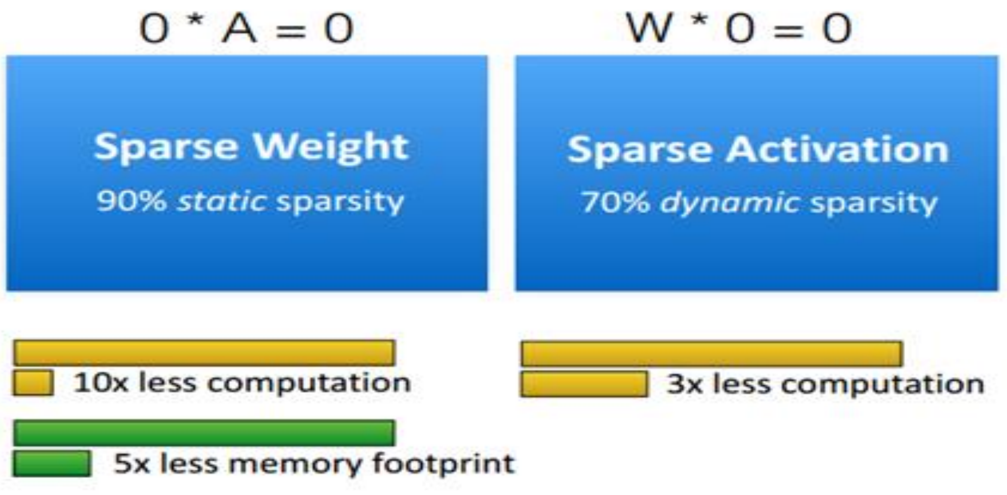


Figure 2-22 EIE Model sparsing

2.3.7 Parallelism

Parallelism means many calculations or the execution of processes are carried out simultaneously, that will decrease the time of execution of CNN architectures. For example, two-dimensional convolution is computed between sliding windows of input feature maps and kernels, and consumes most computation time of CNN as shown in Figure 2-23. Parallelism included inside an output feature map of each layer, known as intra-output parallelism. [15]

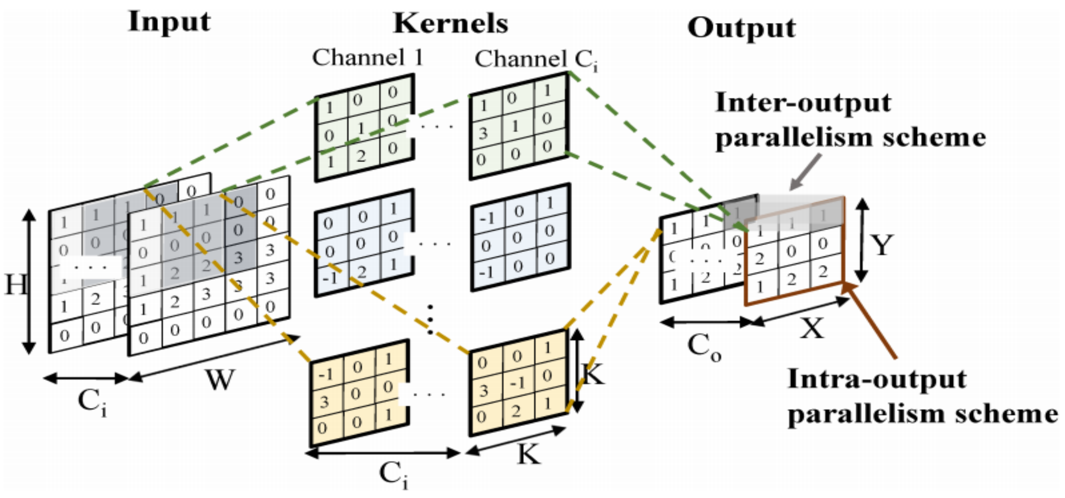

Input
Kernels
Output
Channel 1
Channel C_i
Inter-output parallelism scheme
Intra-output parallelism scheme
H
W
C_i
K
C_o
X
Y

Figure 2-23 Computation of a convolutional layer and its parallelism schemes

2.3.8 Pipelining

The approach is to rearrange the algorithm into a pipeline, where each stage can operate simultaneously with the other stages as shown in Figure 2-24. Pipelining tends to be faster and it can even be more resource efficient.

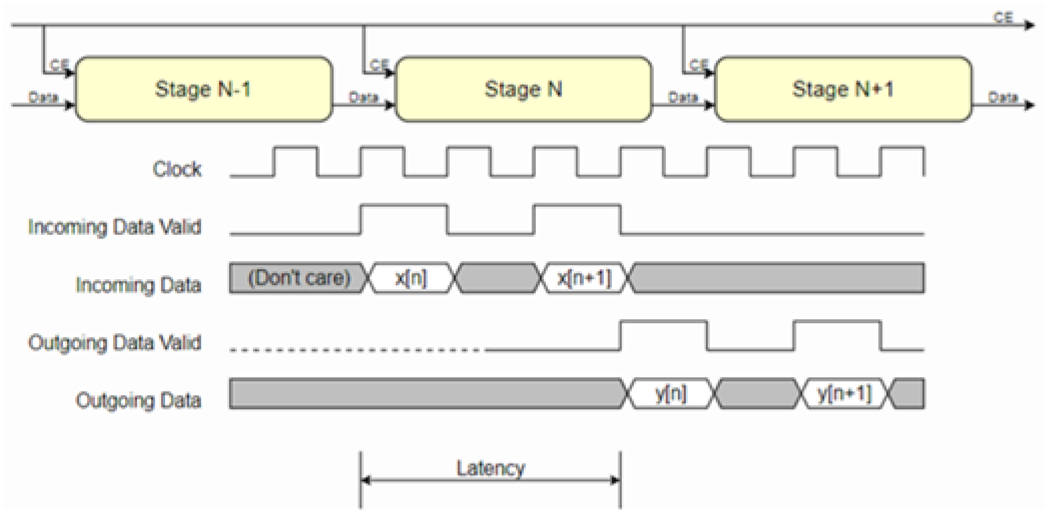


Figure 2-24 Pipelining

2.4 FPGA

2.4.1 Introduction

An FPGA (Field Programmable Gate Array) is an IC consisting of programmable logic gates and interconnections that can be programmed using an HDL (hardware descriptive language) to do a specific function. It is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL), for example VHDL or SystemVerilog.

FPGAs contain an array of programmable logic blocks, and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together", like many logic gates that can be inter-wired in different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR gates. Also, FPGAs contain memory elements, which may be simple flip-flops or more complete blocks of memory. An FPGA can be used to solve any problem which is computable. This is trivially proven by the fact that an FPGA can be used to implement a soft microprocessor.

Most of the digital applications can be implemented with powerful specialized processors, but FPGAs are are sometimes significantly faster for some applications because of their parallel nature and optimality in terms of the number of gates used for a certain process. In all microprocessor-based systems, the functions are executed sequentially, one line of code after another. On the other hand, FPGAs execute their operations in parallel, so FPGAs can be much faster in many applications where speed is crucial. FPGAs also offer great flexibility; the same FPGA IC can be used as a missile guiding system or just a network processing device. As their size, capabilities, and speed increased, they took over additional functions to the point where some are now marketed as full systems on chips (SoC). Particularly with the introduction of dedicated multipliers into FPGA, applications which had traditionally been the sole reserve of DSPs (Digital Signal Processors, a specialized type of processors for signal processing) began to use FPGAs instead. Another trend in the use of FPGAs is hardware acceleration, where one can use the FPGA to accelerate certain parts of an algorithm that need parallel processing and share part of the computation between the FPGA and a generic processor.

2.4.2 FPGA Internal Components

Obviously as shown in Figure 2-25, FPGAs do not only consist of logic gates or look-up tables only. They are made up of many types of logic blocks for implementing many functions and to increase its flexibility.

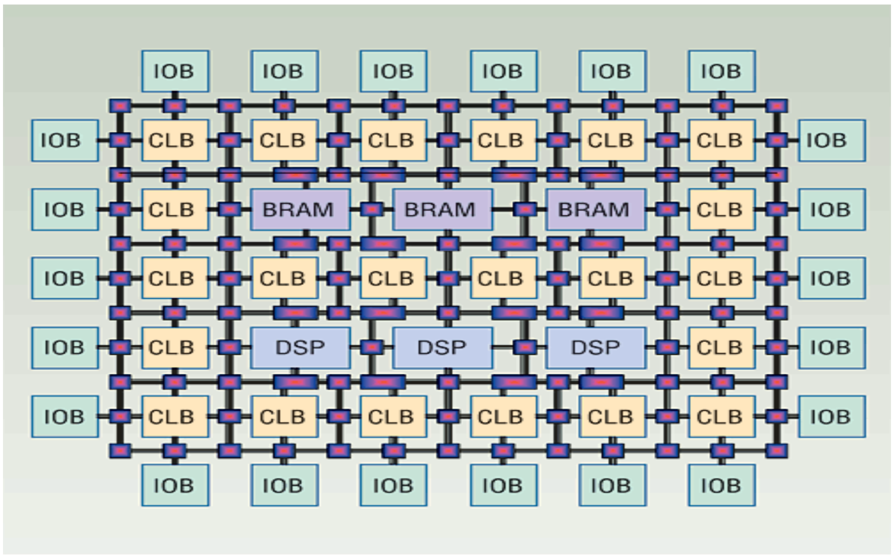


Figure 2-25 FPGA Internal Design

2.4.2.1 Configurable Logic Blocks (CLBs)

These blocks contain the logic for the FPGA. In the dense architecture used by all FPGA vendors today, these CLBs contain enough logic to create a small state machine. The block contains ROMs for creating arbitrary combinatorial logic functions, also known as lookup tables (LUTs). It also contains flip-flops for clocked storage elements, along with multiplexers in order to route the logic within the block and to and from external resources. The multiplexers also allow polarity selection and reset and clear input selection as shown in Figure 2-26.

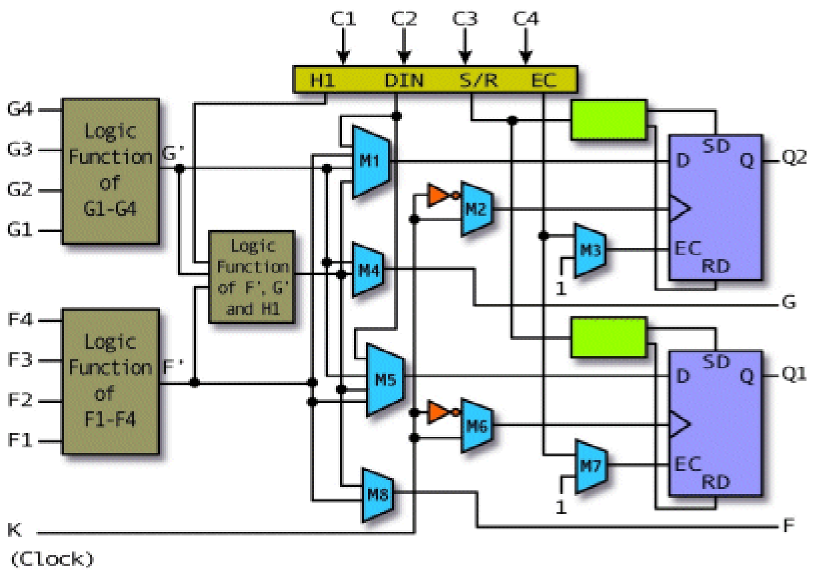


Figure 2-26 FPGA Configurable logic block (CLB)

2.4.2.2 Configurable I/O Blocks

A Configurable input/output (I/O) Block, is used to bring signals onto the chip and send them back off again. 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.

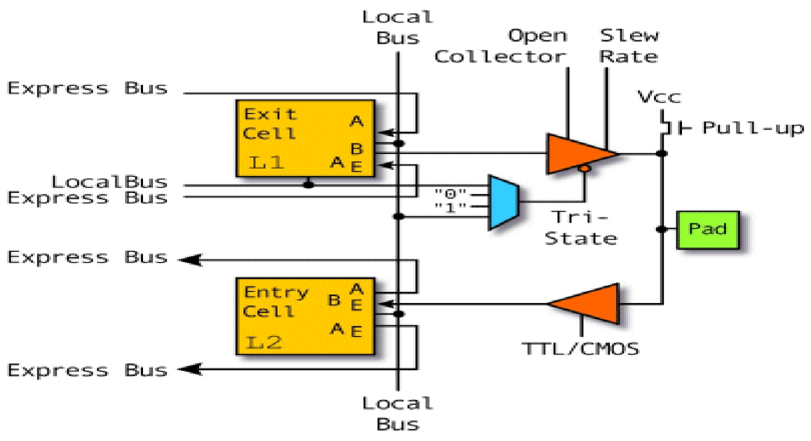


Figure 2-27 FPGA Configurable I/O block

2.4.2.3 Programmable Interconnect

They are the long lines that can be used to connect CLBs to each other on the chip. These lines can also be used as buses within the chip as shown in Figure 2-28. Transistors are used to turn on or off connections between different lines. There are also several programmable switch matrices in the FPGA to connect the long and short lines together in specific, flexible combinations. Special long lines, called global clock lines, are specially designed for low impedance and thus fast propagation times. These are connected to the clock buffers and to each clocked element in each CLB. This is how the clocks are distributed throughout the FPGA, ensuring minimal skew between clock signals arriving at different flip-flops within the chip. In an ASIC, the majority of the delay comes from the logic in the design, because logic is connected with metal lines that exhibit little delay. In an FGPA, however, most of the delay in the chip comes from the interconnect, because the interconnect – like the logic – is fixed on the chip. In order to connect one CLB to another CLB in a different part of the chip often requires a connection through many transistors and switch matrices, each of which introduces extra delay.

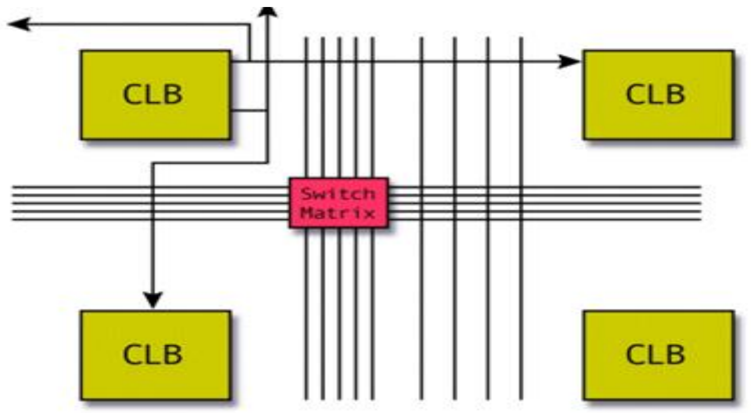


Figure 2-28 FPGA Programmable interconnect

2.4.2.4 Clock Circuitry

Special I/O blocks with special high drive clock buffers, known as clock drivers, are distributed around the chip. These buffers connect to clock input pads and drive the clock signals onto the global clock lines described above. These clock lines are designed for low skew times and fast propagation times. Note that synchronous design is a must with FPGAs, since absolute skew and delay cannot be guaranteed anywhere but on the global clock lines.

2.4.2.5 Block RAM

It is a dedicated RAM block that stores data on the FPGA without consuming any additional LUTs in the design whereas distributed Ram is built up with LUTs. In terms of speed the distributed RAM is faster than Block Rams. It serves as a relatively large memory structure (i.e. larger than distributed RAMs or a bunch of D-Flip-flops grouped together, but much smaller than off chip memory resources).

2.4.2.6 DSP Cores

Digital Signal Processors (DSPs), as shown in Figure 2-29, are another common type of core that is offered as an IP core or an embedded core. These are essentially specialized processors that are used for manipulating analog signals. They are commonly used for filtering and compression of video or audio signals, Multiply-Accumulate block or MAC is implemented as DSP slice and MAC is mainly used as a building block for complex DSP applications.

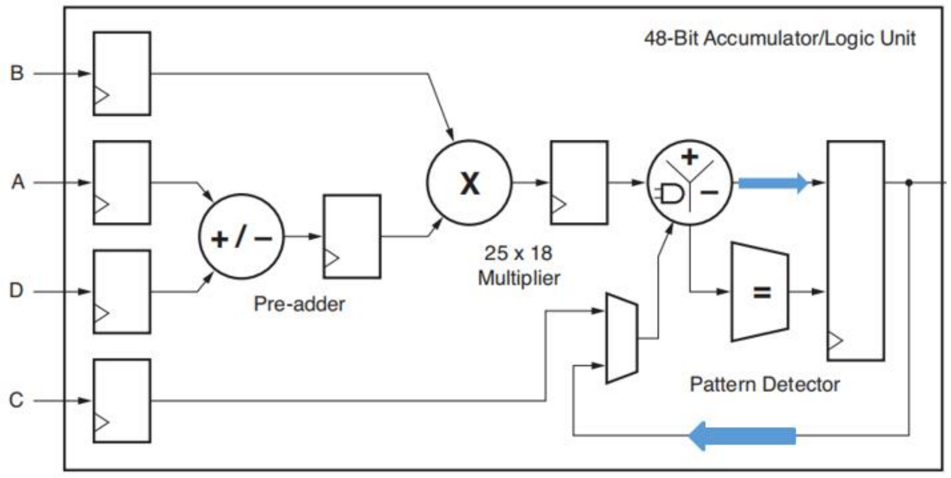


Figure 2-29 DSP Core

2.4.2.7 Embedded Cores

The embedded core will be optimized for the vendor's process to give good timing and power consumption numbers. The core will be placed as a single cell on the silicon die and so the performance of the core will not depend on the rest of the design since it won’t need to be placed and routed.

2.4.2.8 Special I/O Drivers

Special I/O drivers are also being embedded into programmable devices. The newer buses inside personal computers need to have very tightly controlled timing and must be driven by special high-drive, impedance-matched circuits. The I/O buffers need to have inputs with very specific voltage threshold values.

2.4.3 FPGA Design Flow

Figure 2-30 shows the steps of the design flow.

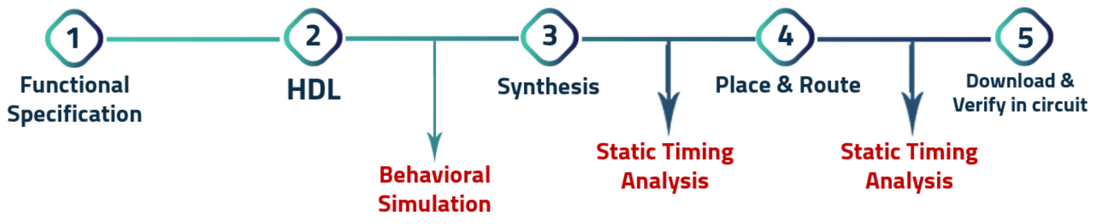


Figure 2-30 FPGA Design Flow

1-Functional Specifications: in this step, all specifications for the application are determined along with good understanding of function of this application**.**

2-HDL: the HDL code that describes that function is written, and then Behavioral Simulation is done to make sure that the HDL describes the function needed correctly.

3-Synthesis: HDL is converted into logic gates and other cells present in the FPGA itself, Static timing analysis is done to approximately calculate the maximum clock delay of the application and calculate the maximum clock speed achieved for the application.

4-Place & Route: The logic blocks and cells in the FPGA are connected together, and Static Timing Analysis is done again to calculate the exact delay model of the application.

5-Download & Verify in circuit: The HDL code is burned on the FPGA

2.5 Summary

This chapter provides background information on Convolution Neural Networks (CNN), discusses the main layers of the CNN (Convolutional Layer, Max Pooling Layer, etc.) with their operations, equations and functions and provides information about different CNN architectures like classic network architectures (LeNet-5 , AlexNet ,VGGNet) or modern network architectures (GoogleNet , ResNet , SqueezeNet), also it discusses several methods to improve the efficiency of deep learning implementation, then it describes the training process with both forward and backward propagation and indicates how it minimizes the loss and improve the accuracy. Finally, it discusses background on FPGAs, including a brief overview of FPGA internal components and design flow.

Chapter 3: ZynqNet

The chosen CNN architecture for implementation on FPGA is ZynqNet, due to having low computational complexity and low number of parameters that needs to be stored; in this chapter ZynqNet will be overviewed and discussed from the software point of view.

3.1 Overview

ZynqNet CNN is a stripped-down version of SqueezeNet and consists exclusively of convolutional layers, ReLU nonlinearities and a global average pooling, it had top-1 and top-5 error rate of 41.52% and 15.4% respectively on the test data of image-Net dataset which was better than SqueezeNet error rates, also number of MACC operations and total number of activations reduced by 38% and 40% respectively with regard to the original SqueezeNet.

The neural network developed by David Gschwend in August 2016, ETH Zurich which has 2.5 million parameters, consists of 27 convolutional layers and only one average pooling layer

3.2 ZynqNet vs. Other ConvNets

- The computational complexity of ZynqNet has been lowered by 38% in comparison to the original SqueezeNet and by more than 50% compared to AlexNet as shown in Figures 3-1 and 3-2. [2]
- The number of parameters is roughly twice as SqueezeNet parameters, but it’s still roughly an order of magnitude less than most. [2]
- Top-5 error is better than AlexNet and SqueezeNet.
- It is one of the least architectures in terms of number of MACCs.

	#conv. layers	#MACCs [millions]	#params [millions]	#activations [millions]	ImageNet top-5 error
ZynqNet CNN	**18**	**530**	**2.5**	**8.8**	**15.4%**
AlexNet	5	1 140	62.4	2.4	19.7%
Network-in-Network	12	1 100	7.6	4.0	~19.0%
VGG-16	16	15 470	138.3	29.0	8.1%
GoogLeNet	22	1 600	7.0	10.4	9.2%
ResNet-50	50	3 870	25.6	46.9	7.0%
Inception v3	48	5 710	23.8	32.6	5.6%
Inception-ResNet-v2	96	9 210	31.6	74.5	4.9%
SqueezeNet	18	860	1.2	12.7	19.7%
SqueezeNet v1.1	18	390	1.2	7.8	19.7%

Figure 3-1 Comparison of ZynqNet to CNN architectures

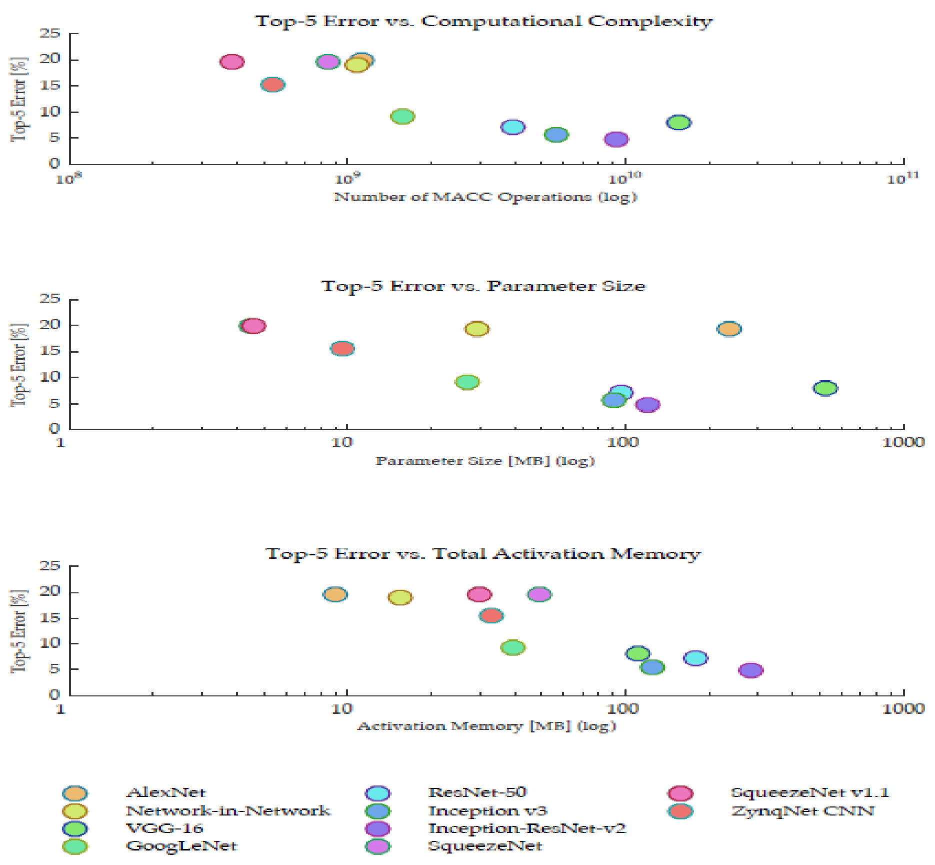


Figure 3-2 Comparison graphs

3.3 ZynqNet Architecture

ZynqNet CNN consists of 27 layers and 2.5 million parameters; it consists of convolutional layers, ReLU nonlinearities and a global average pooling as shown in Figure 3-3, detailed description of layers and parameters are given in Table 3-1.

The computational complexity in ZynqNet comes almost entirely from the 1×1 and 3×3 convolutions, which add up to 530 million MACC operations. The ReLU nonlinearities amount to 3 million comparisons. The average pooling requires 66,000 additions and one division. ZynqNet was trained on ImageNet classification training set which contained 1.28 million high-resolution images to classify 1000 different classes

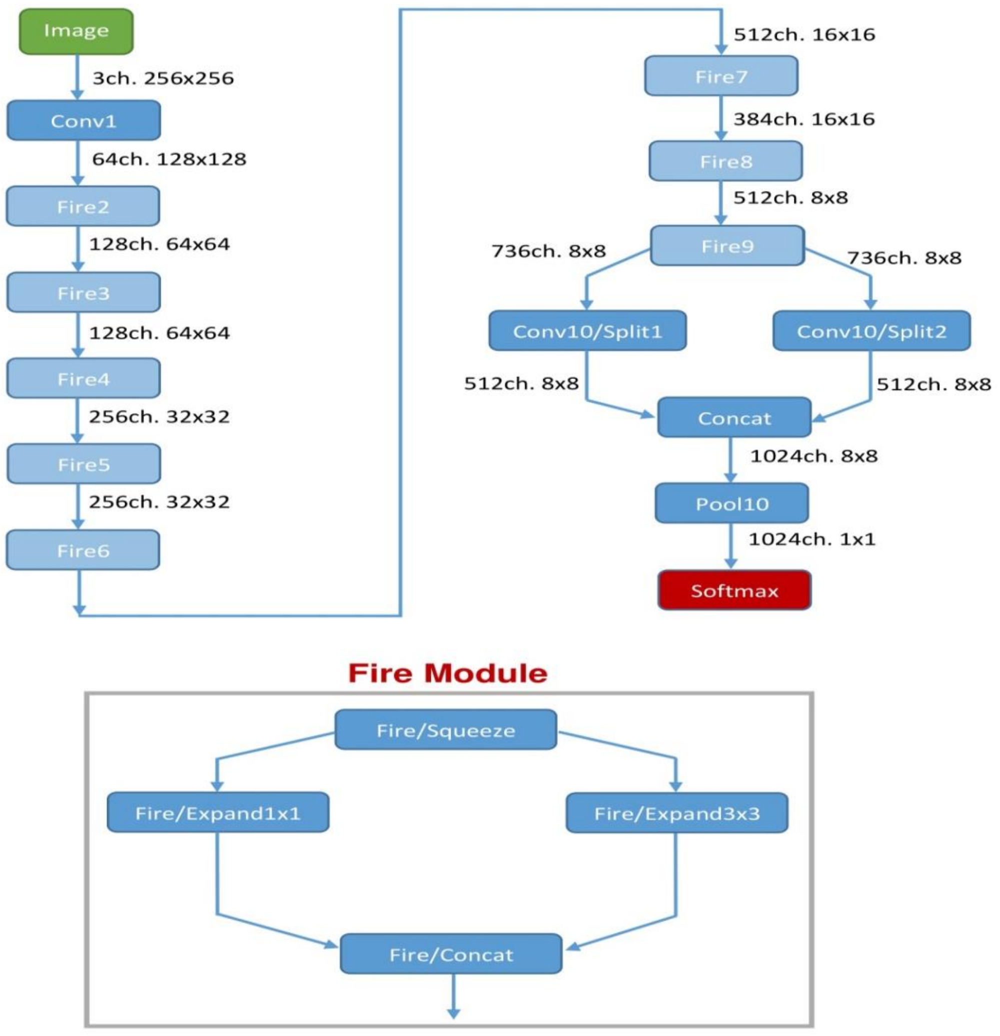


Figure 3-3 ZynqNet Architecture

Table 3-1 Detailed Description of all ZynqNet CNN Layers and their Parameters

ID	Layer Name	Layer Type	Kernel	Stride	Pad	CH in	W x H in	CH out	W x H out
1	Conv1	Convolution	3x3	2	1		256x256	64	128x128
2	Fire2/Squeeze3x3	Convolution	3x3	2	1	64	128x128	16	64x64
3	Fire2/Expand1x1	Convolution	1x1	1	0	16	64x64	64	64x64
4	Fire2/Expand3x3	Convolution	3x3	1	1	16	64x64	64	64x64
5	Fire2/Concat	Concat				128	64x64	128	64x64
6	Fire3/Squeeze1x1	Convolution	1x1	1	0	128	64x64	16	64x64
7	Fire3/Expand1x1	Convolution	1x1	1	0	16	64x64	64	64x64
8	Fire3/Expand3x3	Convolution	3x3	1	1	16	64x64	64	64x64
9	Fire3/Concat	Concat				128	64x64	128	64x64
10	Fire4/Squeeze3x3	Convolution	3x3	2	1	128	64x64	32	32x32
11	Fire4/Expand1x1	Convolution	1x1	1	0	32	32x32	128	32x32
12	Fire4/Expand3x3	Convolution	3x3	1	1	32	32x32	128	32x32
13	Fire4/Concat	Concat				256	32x32	256	32x32
14	Fire5/Squeeze1x1	Convolution	1x1	1	0	256	32x32	32	32x32
15	Fire5/Expand1x1	Convolution	1x1	1	0	32	32x32	128	32x32
16	Fire5/Expand3x3	Convolution	3x3	1	1	32	32x32	128	32x32
17	Fire5/Concat	Concat				256	32x32	256	32x32
18	Fire6/Squeeze3x3	Convolution	3x3	2	1	256	32x32	64	16x16
19	Fire6/Expand1x1	Convolution	1x1	1	0	64	16x16	256	16x16
20	Fire6/Expand3x3	Convolution	3x3	1	1	64	16x16	256	16x16
21	Fire6/Concat	Concat				512	16x16	512	16x16
22	Fire7/Squeeze1x1	Convolution	1x1	1	0	512	16x16	64	16x16
23	Fire7/Expand1x1	Convolution	1x1	1	0	64	16x16	192	16x16
24	Fire7/Expand3x3	Convolution	3x3	1	1	64	16x16	192	16x16
25	Fire7/Concat	Concat				384	16x16	384	16x16
26	Fire8/Squeeze3x3	Convolution	3x3	2	1	384	16x16	112	8x8
27	Fire8/Expand1x1	Convolution	1x1	1	0	112	8x8	256	8x8
28	Fire8/Expand3x3	Convolution	3x3	1	1	112	8x8	256	8x8
29	Fire8/Concat	Concat				512	8x8	512	8x8
30	Fire9/Squeeze1x1	Convolution	1x1	1	0	512	8x8	112	8x8
31	Fire9/Expand1x1	Convolution	1x1	1	0	112	8x8	368	8x8
32	Fire9/Expand3x3	Convolution	3x3	1	1	112	8x8	368	8x8
33	Fire9/Concat	Concat				736	8x8	736	8x8
34	Conv10/Split1	Convolution	1x1	1	0	736	8x8	512	8x8
35	Conv10/Split2	Convolution	1x1	1	0	736	8x8	512	8x8
36	Conv10	Concat				1024	8x8	1024	8x8
37	Pool10	Pooling	8x8			1024	8x8	1024	1x1
38	Loss	Softmax				1024	1x1	1024	1x1

3.4 Convolutional Layers

In ZynqNet the convolutional layers are (Conv1, Fire_n/Squeeze, Fire_n/Expand1x1, Fire_n/Expand3x3, Conv10/Split1 , Conv10/Split2) where n could be a number from 2 to 9 as shown in Table 3-1. The convolutional operation is done by applying ($ch_{in}x\ ch_{out}$) filters of size (k x k) to generate the output feature maps. As the filter is sliding, or convolving, around the input image, it is multiplying the values in the filter with the original pixel values of the image (i.e.. computing element wise multiplications). These multiplications are all summed up to produce a single output. The process is repeated for every location on the input volume as shown in Figure 3-4. For filters larger than 1x1, border effects reduce the output dimensions. To avoid this effect, the input image is padded with p = $\frac{k}{2}$ zeros on each side. This effect can be applied with stride of s, which reduces the output dimensions to $w_{out} = \frac{w_{in}}{s}$, $h_{out} = \frac{h_{in}}{s}$. [2]

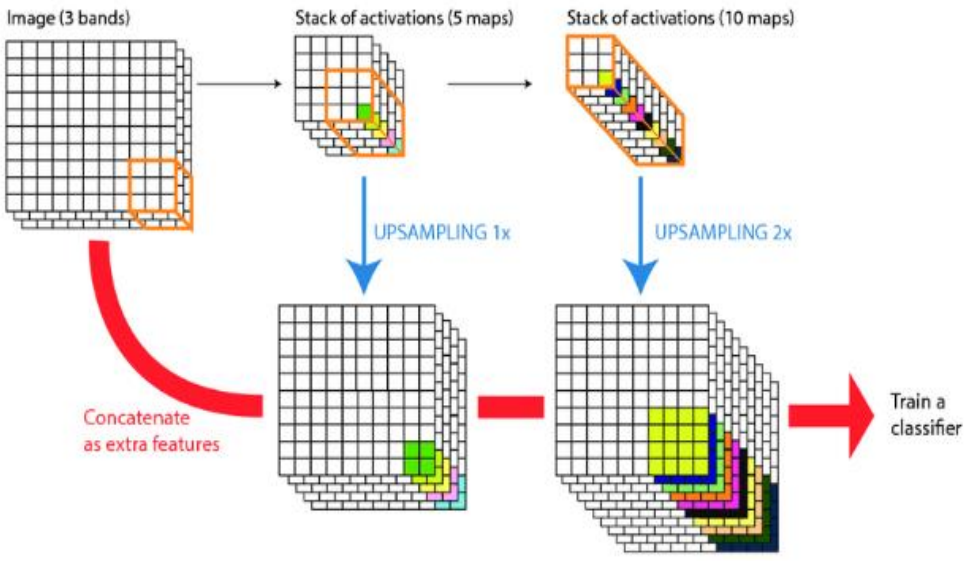


Figure 3-4 Convolutional Layer

The Conv layer's parameters consist of a set of learnable filters. Every filter is small spatially (along width and height) but extends through the full depth of the input volume. Each of these filters can be thought of as feature identifiers (edges, simple colors, and curves). First layer filters detect low level features such as edges and curves. In order to predict whether an image is a type of object, the network must be able to recognize higher level features. To extract high level features the output of the first layer is applied to a set of filters (pass it through the 2nd Conv layer). As the network get deeper and go through more Conv layers, activation maps can represent more complex features.

In practice, a CNN learns the values of these filters on its own during the training process. However, other parameters are still need to be specified such as number of filters, filter size, architecture of the network before the training process.

3.4.1 Stride

Stride and padding are two main parameters that can be changed to modify the behavior of each layer. Stride controls how the filter convolves around the input volume. As shown in Figure 3-5, the filter convolves around the input volume by shifting one unit at a time. The amount by which the filter shifts is the stride. In that case, the stride was implicitly set at 1. Stride is normally set in a way so that the output volume is an integer and not a fraction. It is assumed to have a 7 x 7 input volume, a 3 x 3 filter and a stride of 1. The output will be as shown in Figure 3-5. [16]

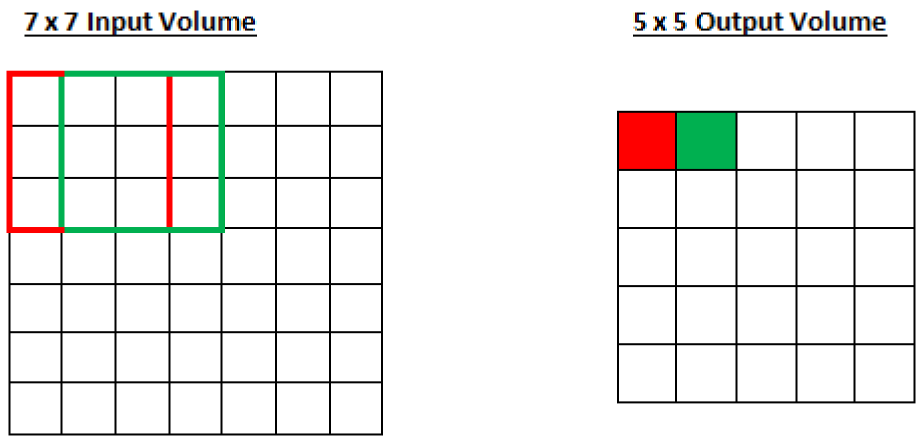


Figure 3-5 Stride of 1

For the same example but with a stride of 2, the output will be as shown in Figure 3-6.

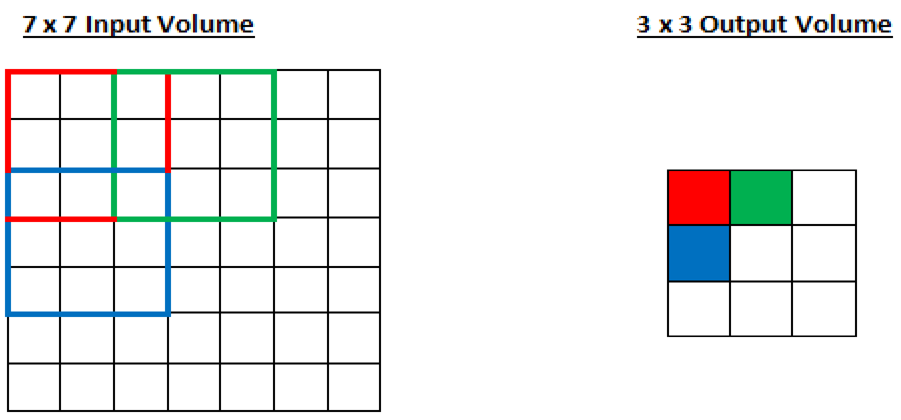


Figure 3-6 Stride of 2

So, as noticed, the receptive field is shifting by 2 units now and the output volume shrinks as well. Notice that if stride is set to 3, then there will be issues with spacing and making sure the receptive fields fit on the input volume. Normally, programmers will increase the stride if they want receptive fields to overlap less and if they want smaller spatial dimensions.

3.4.2 Padding

It is assumed to have a three 5 x 5 x 3 filters to a 32 x 32 x 3 input volume, then by convolution, the output volume would be 28 x 28 x 3, it is noticed that the spatial dimensions decrease. As Conv layers are applied, the size of the volume will decrease faster than needed. In the early layers of the network, it's good to preserve as much information about the original input volume so that low-level features can be extracted. It is assumed that the same Conv layer is applied but the output volume has to remain 32 x 32 x 3. To do this, a zero padding of size 2 can be applied to that layer. Zero padding pads the input volume with zeros around the border. If padding of 2 is used, then this would result in a 36 x 36 x 3 input volume as shown in Figure 3-7. [16]

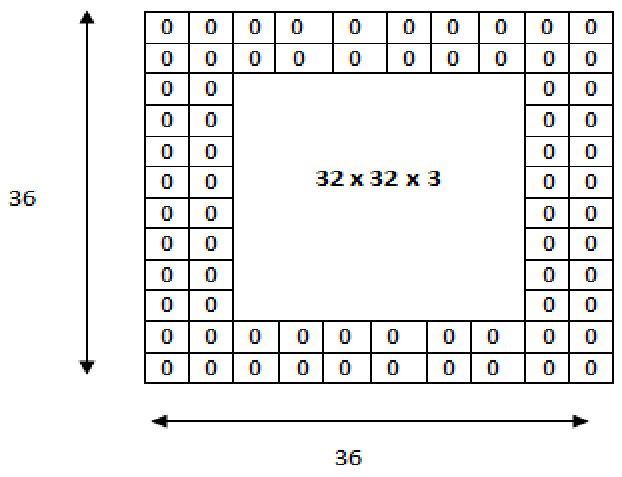


Figure 3-7 Padding of 2

If a stride of 1 is used and if the size of zero padding is set to $\frac{K-1}{2}$ where K is the filter size, then the input and output volume will always have the same spatial dimensions. The formula for calculating the output size for any given conv layer is $O = \frac{(W-K+2P)}{S} + 1$ Where O is the output height/length, W is the input height/length, K is the filter size, P is the padding, and S is the stride.

3.4.3 ReLU

After each conv layer, it is convention to apply a nonlinear layer (or activation layer) immediately afterward. The purpose of this layer is to introduce nonlinearity to a system that basically has just been computing linear operations during the conv layers (just element wise multiplications and summations).In the past, nonlinear functions like tanh and sigmoid were used, but researchers found out that ReLU layers work far better because the network is able to train a lot faster (because of the computational efficiency) without making a significant difference to the accuracy. It also helps to alleviate the vanishing gradient problem, which is the issue where the lower layers of the network train very slowly because the gradient decreases exponentially through the layers. The ReLU layer applies the function f(x) = max(0, x) to all of the values in the input volume as shown in Figure 3-8. In basic terms, this layer just changes all the negative activations to 0. This layer increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the conv layer. [16]

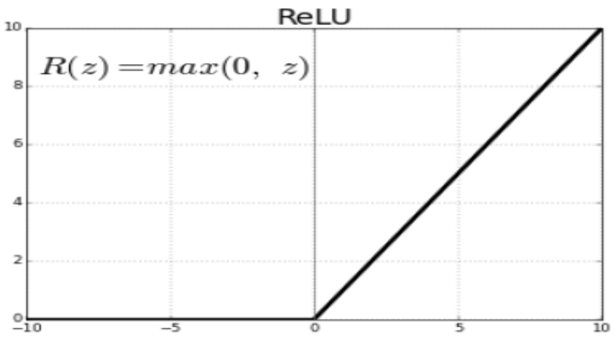


Figure 3-8 ReLU Function

3.4.4 Fire Module

Fire module is the basic building block of ZynqNet as it contains 8 stacked fire modules, each fire module contains: a squeeze convolution layer feeding into an expand layer that has a mix of 1×1 and 3×3 convolution filters as in Figure 3-9, the output channels of expand layer are concatenated to form a single feature map.

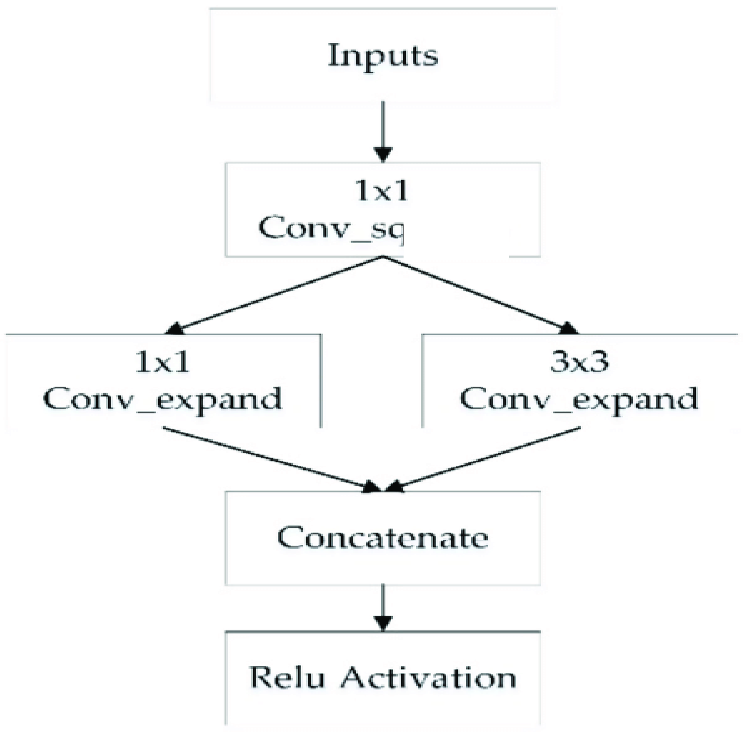


Figure 3-9 Fire Module

Squeeze layer decreases the number of input channels to 3x3 filters to decrease number of computations made and decrease the number of parameters, then expand layer increase the number of channels again and generate feature maps.

Maximum pooling used in SqueezeNet is removed and instead in ZynqNet stride of 2 can be used subsequent convolutional layer but as these layers are 1x1 squeeze layers, some important information may be lost so the kernel size in these squeeze layers was changed to 3x3 with stride of 2, this modification results in 12% more parameters and 18% more MACC operations but also results in 1.5% increase in accuracy.

3.4.5 Pooling Layer

Pooling layers provide an approach to down sampling feature maps by summarizing the presence of features in patches of the feature map. Two common pooling methods are average pooling and max pooling which summarize the average presence of a feature and the most activated presence of a feature respectively as shown in Figure 3-10.

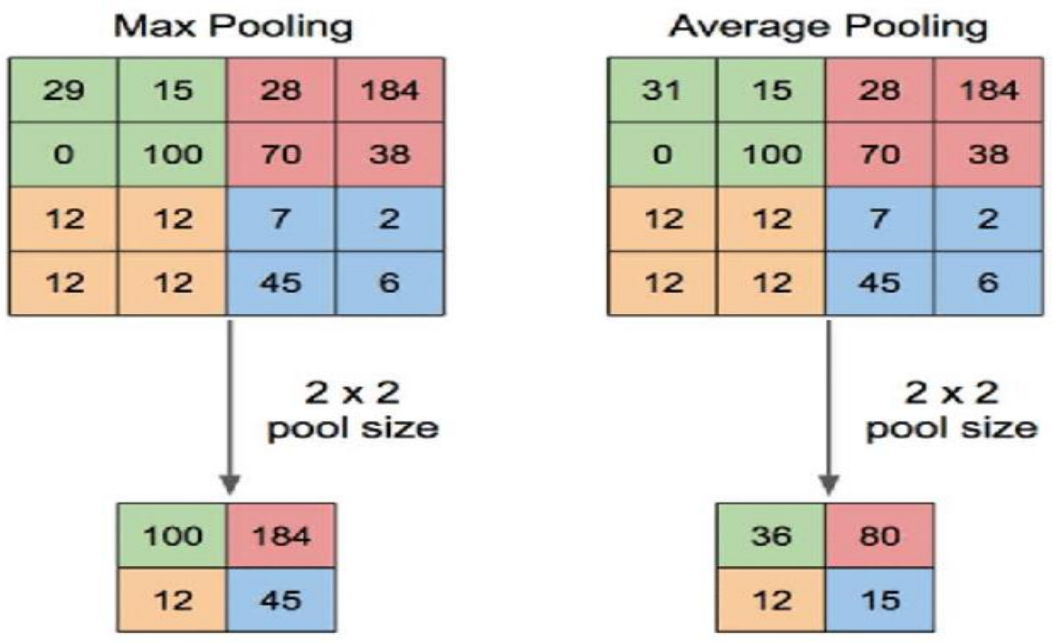


Figure 3-10 Maximum and Average Pooling Layers

Global average pooling is used in ZynqNet after last convolutional layer; it reduces the spatial dimensions from 8×8 pixels to 1×1 pixel by computing the mean.

3.4.6 Dropout

Dropout exists only during training of the model, so it's not an implemented layer during inference. Dropout are a popular method to combat over fitting in large CNNs. They are used to randomly drop a selectable percentage of their connections during training, which prevents the network from learning very precise mappings, and forces some abstraction and redundancy to be built into the learned weights as shown in Figure 3-11. [2]

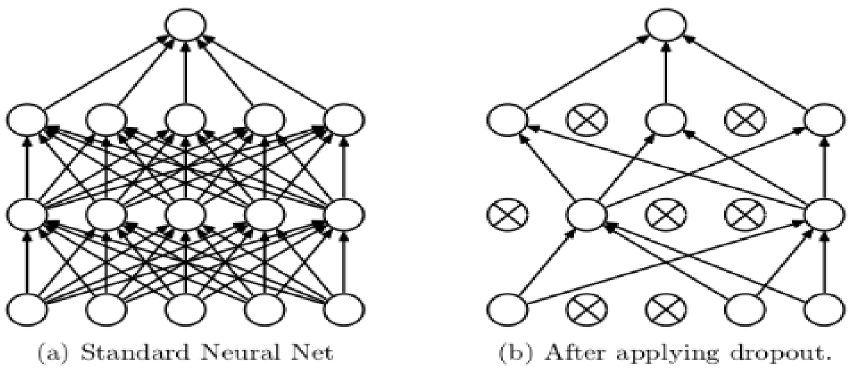


Figure 3-11 Dropout Layer

3.4.7 Softmax

Softmax layers are the most common classifiers. A classifier layer is added behind the last convolutional or fully-connected layer in each image classification CNN, and squashes the raw class scores $\boldsymbol{z_i}$ into class probabilities $\boldsymbol{p_i}$ according to $\boldsymbol{p_i} = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$, which results in a vector P that sums up to 1 as shown in Figure 3-12. [2]

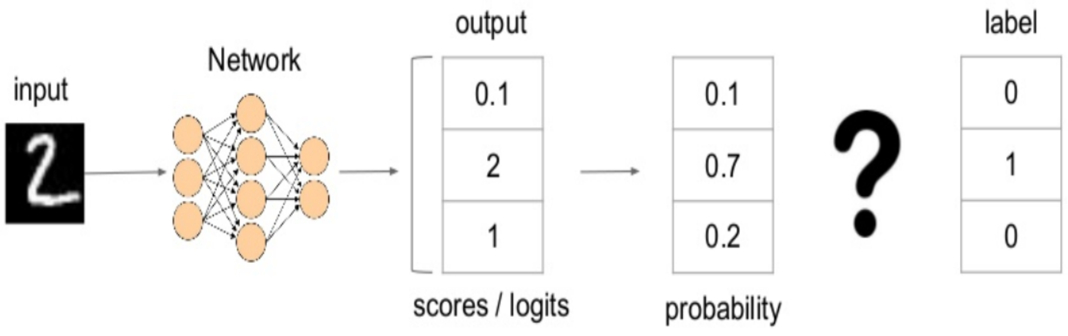


Figure 3-12 Softmax Layer

3.5 ZynqNet Optimizations

3.5.1 Large filter input layer approximation

Usually the convolutional input layer has large kernel size and a large stride. The advantage of that is to make a large reception of the network. But on the other hand, it costs a very expensive hardware. As in SqueezeNet a 7×7 filter requires 5.4× more MACC operations than a 3×3 filter. As long as the learned filters are well-behaved, a 7×7 kernel can be approximated by three stacked 3×3 filters as shown in Figure 3-13, that need only 27/49 of the computations as shown in Figure 3-14 and have the same receptive field. This optimization causes less than 1% drop in the accuracy. [2]

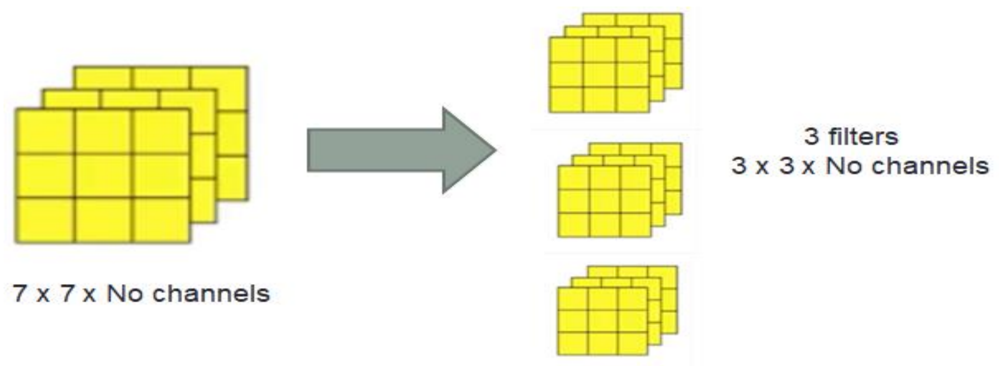


Figure 3-13 Large filter approximation in ZynqNet

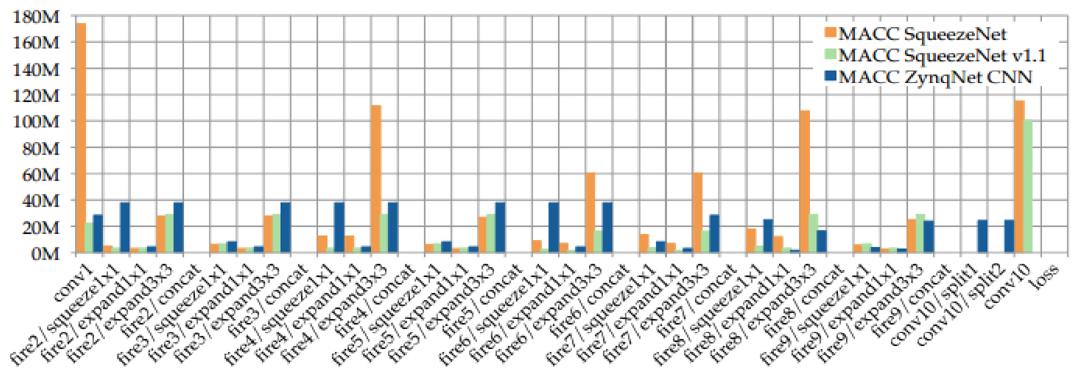


Figure 3-14 Number of MACC Operations for SqueezeNet, SqueezeNet v1.1 and ZynqNet CNN

3.5.2 Unnecessary Padding

Padding makes no sense for 1×1 filters, and setting pad=0 will save some of the MACC cycles. The original SqueezeNet used pad=1 in the 1×1 conv layer Conv10, which is removed in ZynqNet. [2]

3.5.3 Out-of-Sync Dimension Adjustments

The spatial output dimensions in are periodically stepped down (using stride 2 in layers Conv1, Fire2, Fire4, Fire8, and using global pooling in Pool10). The number of output channels is periodically increased. But in SqueezeNet the spatial shrinking and the channel-wise expansion are not ideally synchronized (and fire4 as well as fire8 increase the number of output channels before decreasing the pixel count) leading to a surge in computational complexity. ZynqNet solve this problem. The modification saves up to 40 % in activation memory and reduces the computational complexity in fire4 and fire8 by a factor of 3.7 and 3.9 respectively. [2]

3.5.4 Layer splitting

To make the large layers fit onto the FPGA, they have been splitted into two parallel convolutional layers then concatenated along the channel dimension. As shown in Figure 3-15 in ZynqNet, the fully connected layer Conv10 layer has input channels (chin) = 736 and output channels chout = 1024 and would therefore require n = chin * chout = 753664 kernels of size 1×1. it has been split into two parallel convolutional layers Conv10/split1 and Conv10/split2 with chout = 512. [2]

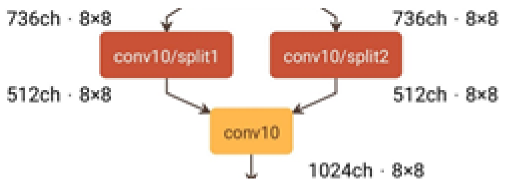


Figure 3-15 Conv10 layer in ZynqNet

3.5.5 Equalization of Layer Capacities

The mean concept of CNN is to transform a large amount of pixels with low individual information density into very few outputs of high abstraction level. The layer capacity *w*out × *h*out × *ch*out can be seen as a measure for this concentration of information. The layer capacities of SqueezeNet, SqueezeNet v1.1 and ZynqNet all converge from more than one million data points to just 1000 class probabilities as shown in Figure 3-16. Further, both SqueezeNet versions have a strong peak in Conv10. ZynqNet CNN follows a much smoother and more regular capacity reduction, which saves resources, but also increases accuracy by almost 2.3%. [2]

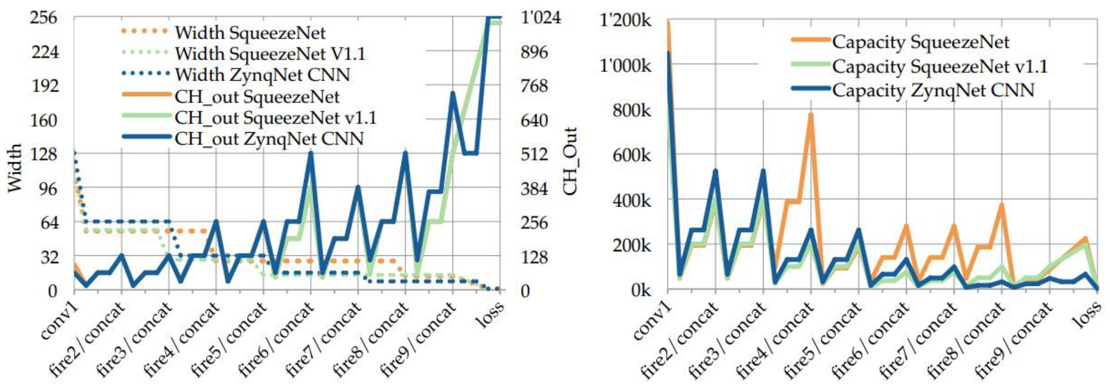


Figure 3-16 Per-Layer Dimension Analysis of SqueezeNet, SqueezeNet v1.1 and ZynqNet CNN

3.5.6 Fine-Tuning

Using fine tuning in final experiments means re-training the finalized network for a few epochs with a very low learning rate, sometimes a slightly better optimum can be reached. However, the result is 0.2% accuracy gain. [2]

3.6 Software Accuracy

ZynqNet CNN was tested on ImageNet validation set using python code; the trained model weights were extracted from the Caffe model made for ZynqNet, the python code converts the input picture from (RGB) representation to Caffe representation (BGR) and provides the preprocessing required for the input image and the implementation of each layer in the network.

The accuracy was measured on ImageNet validation set [17] for two different number of bits representation of the data.

- For the ideal representation of the data using 64 double data type the accuracy was 58.48%.
- For 16-bit fixed point with:
  - 1-bit for integer,1-bit for sign bit and 14-bits for fraction part for weights
  - 14-bit integar,1-bit sign and 1-bit fraction for convolutional output

The accuracy reached 57.49%.

3.7 Preparing data for Hardware implementation

To prepare the input data for the hardware simulation, the data was required to fit into ROM, therefore the input data is unrolled from 3-D (BGR) representation to the 1-D representation to fit into the input ROM as shown in Figure 3-17.

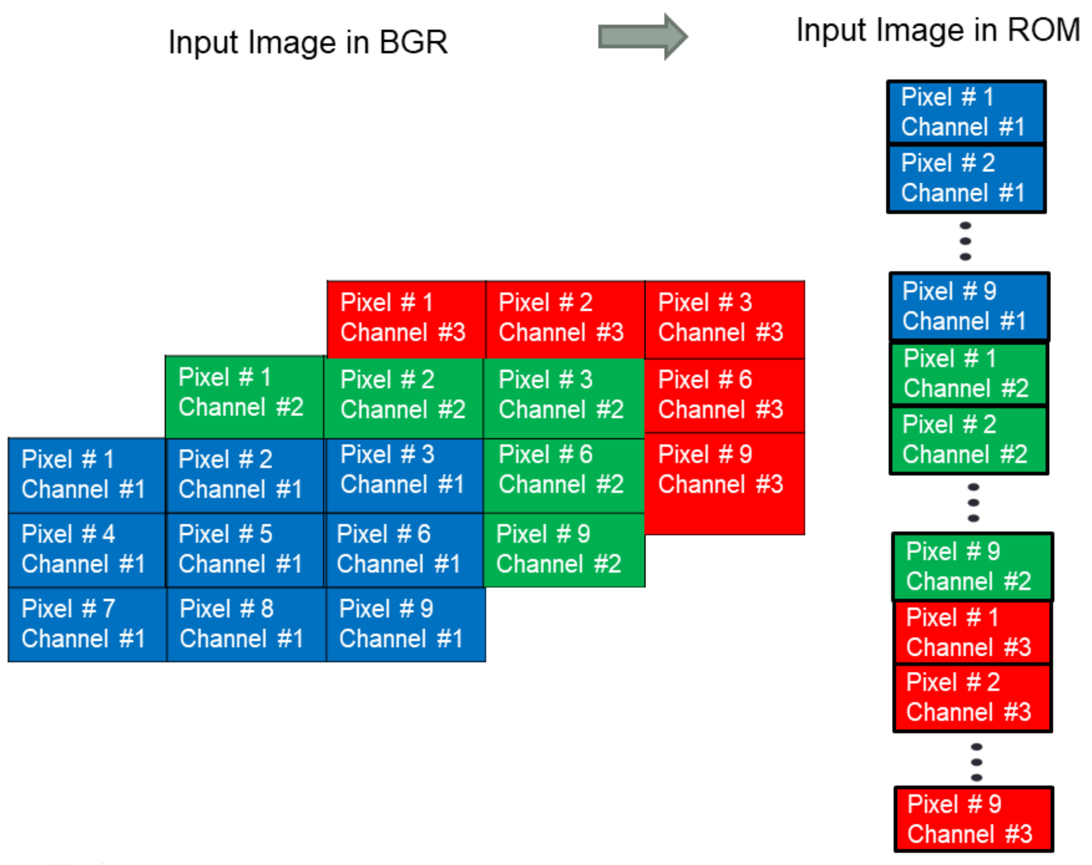


Figure 3-17 Unrolling the input data from 3-D to 1-D

3.8 Fixed Point Background

Deep convolutional neural network (CNN) inference requires significant amount of memory and computation, which limits its deployment on embedded devices. To alleviate these problems to some extent, prior research utilize low precision fixed-point numbers to represent the CNN weights and activations. However, the minimum required data precision of fixed-point weights varies across different networks and also across different layers of the same network.

A fixed-point representation of a number consists of integer and fractional components and sign bit as shown in Figure 3-18, where WL represents word length, S represents the sign bit, I represent the integer bits and F represents the fractional bits. With this representation the range of numbers is $[2^{-I}\ , 2^{I}]$, and a step size (resolution) of 2^{-F} . [4]

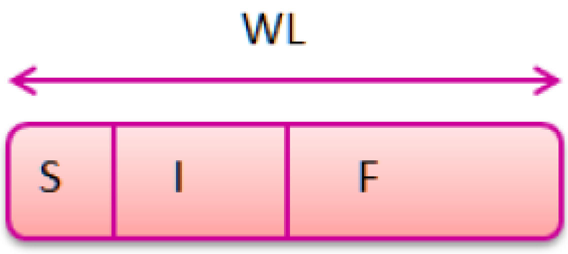


Figure 3-18 Fixed point data representation

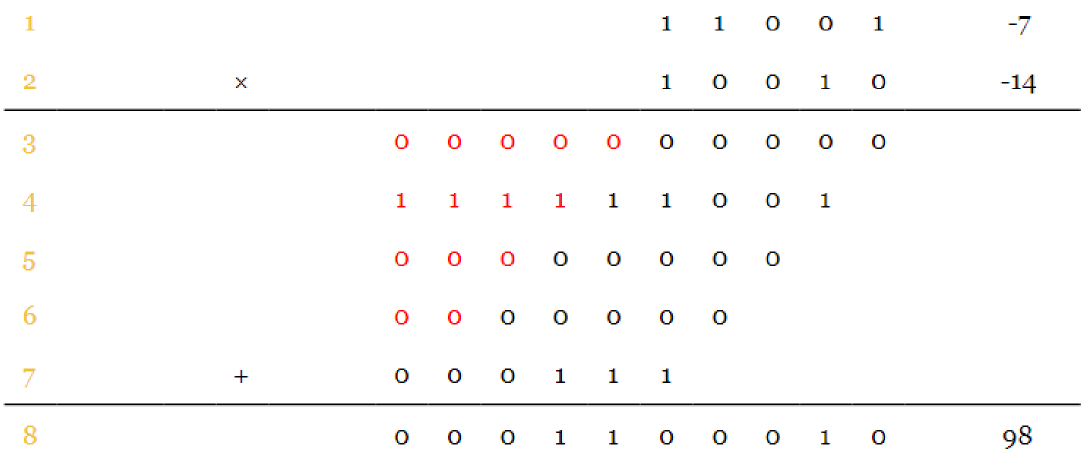
3.8.1 Fixed Point Multiplication

Fixed-point multiplication is the same as 2's compliment multiplication but requires the position of the "point" to be determined after the multiplication to interpret the correct result. The determination of the "point's" position is a design task. The actual implementation does not know (or care) where the "point" is located. This is true because the fixed-point multiplication is exactly the same as a 2's complemented multiplication, no special hardware is required. Consider the following illustrative example assuming a has WL = 5, I=1, F=3 and S=1, and b has WL = 5, I=1, F=3 and S=1,

$$a = 11.001_2 = -0.875|_{\text{decimal}} = 11001|_{\text{fixed point representation}}$$

$$b = 10.010_2 = -1.75|_{\text{decimal}} = 10010|_{\text{fixed point representation}}$$

The output product will be as shown in Figure 3-19.



1							1	1	0	0	1	-7
2	×						1	0	0	1	0	-14
3		0	0	0	0	0	0	0	0	0	0	
4		1	1	1	1	1	1	0	0	1		
5		0	0	0	0	0	0	0	0			
6		0	0	0	0	0	0	0				
7	+	0	0	0	1	1	1					
8		0	0	0	1	1	0	0	0	1	0	98

Figure 3-19 Fixed point multiplication

The output = $0001100010|_{\textbf{fixed point representation}} = 0001.100010_2 = 1.53125|_{\textbf{decimal}}$

The number of bits required for the product (result) is a's WL + b's WL. Note that the fractional bits in the product are equal to a’s F + b’s F.

3.8.2 Fixed Point Addition

Performing the addition using the same numbers from the multiplication problem,

$$\begin{array}{l} 11.001 \\ \underline{10.010} \end{array}$$

$101.011 = -2.625|_{\textbf{decimal}}$

When adding (subtracting) two numbers an additional bit is required for the result. When adding more than two numbers all of the same WL width, the number of bits required for the result is $WL{=}WL{+}log2\ (N\ x\ WL) + log2\ (N)$, where N is the number of elements being summed.

3.9 Summary

This chapter provides information on the chosen CNN which is ZynqNet, having a quick overview on its accuracy for the top-5 and top-1 results on ImageNet validation sets, the number of its layers and their arrangement, and the number of parameters needed by ZynqNet. It also shows the effect of changing the number of bits of the fixed point data propagating between layers on the accuracy and the chosen number of bits based on its effect on accuracy and compromising this effect with the number of resources utilized by the design.

Chapter 4: Hardware Methodology

4.1 ZynqNet Design

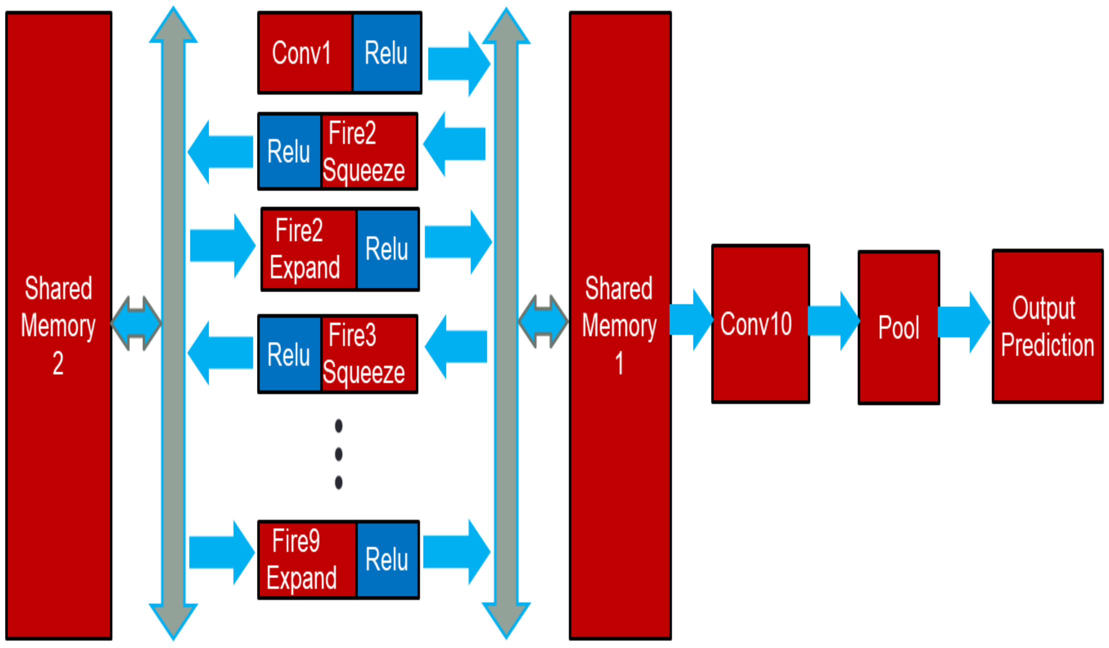


Figure 4-1 ZynqNet block diagram

The proposed approach as in Figure 4-1 is mainly based on designing each convolution layer separately with their separate weights storage and Multiply Accumulation (MAC) units needed for performing the convolution operation. Two shared memories are used to store the intermediate storage between layers which are Conv1 and all Fire layers, Conv10 is interleaved with one global average pooling layer to reduce the dimension of features retaining the most dominant information. Finally Output Prediction block is implemented to get the index of the prediction of the model.

4.2 Convolution Layer Implementation

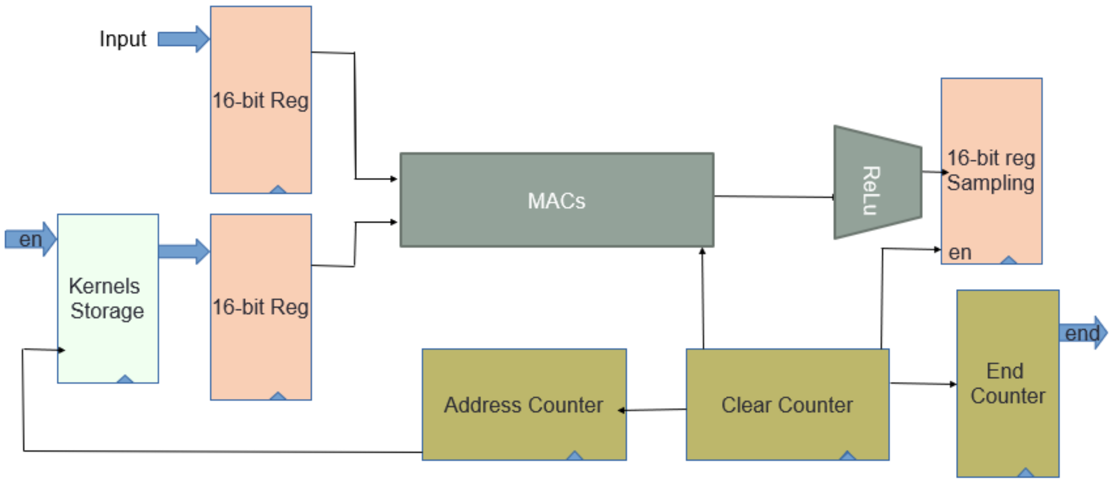


Figure 4-2 Convolution layer block diagram

In ZynqNet implementation, each layer contains specific number of MACs and it's chosen to work with full duty (number of MACs = number of output filters). Also, each layer has its own enable signal as well as end flag. Separate weights storage is generated for each layer. The convolution operation is performed using Multiply and Accumulation units, and then it gets through the ReLU function and quantization operation. A clear counter is used to clear the accumulation register in MAC after a window is convolved and determine the weights storage offset. It also determines when the output is ready to be sampled as shown in Figure 4-2.

4.2.1 MAC

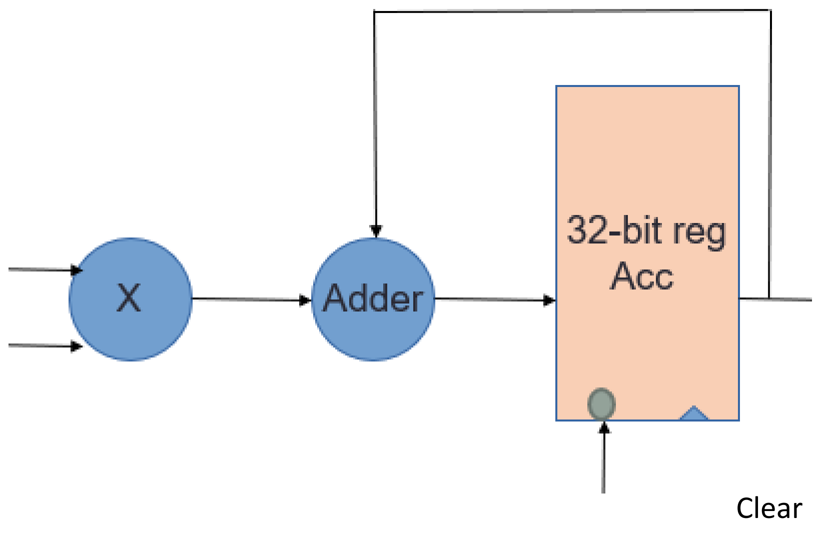


Figure 4-3 MAC implementation

Figure 4-3 shows internal design of one MAC (which corresponds to one DSP) that consists of 16x16 multiplier, 32-bit accumulation register, 32-bit adder and clear signal used to clear register after each output, i.e. after (Kernel x Kernel x CHin) cycles.

The number of MACs or DSPs per layer depends on the number of output filters, DSPs dedication for all filters results in 4448 DSP, which is unfeasible by Virtex-7 Series DSPs that is equal to 3600. A solution for this problem is to make identical layers shared in DSPs, like fire2/expand3x3 and fire3/expand3x3 layers, the same 64 DSPs can be used for the two layers.

4.2.2 Kernel Storage

4.2.2.1 Weights and Bias Implementation

ZynqNet holds around 2.5 M parameters, of which are network weights and bias. The relatively low parameters make it feasible to implement the architecture on FPGAs for acceleration approach. However, this architecture deployment requires more attention to weights and bias rather than all other building blocks, for they consume a huge number of resources and, of course, they affect speed directly. Usually, fixed parameters are stored into Read Only Memories (ROMs), with a controller to take over fetches, flags and more.

4.2.2.2 FPGAs Read Only Memory

Some FPGAs have off-chip ROMs, which have great utility in account of high latency, however, ZynqNet implementation targets speed as its first priority, so, off-chip memories would not be an option. FPGAs hold two types of memories, any of which could fit for ZynqNet parameters.

4.2.2.2.1 BRAMs

FPGAs normally have on-chip BRAM matrix, which could be configured as FIFO, RAM or ROM. Targeted device (Virtex-7 x690t) contains a sum of 2940 BRAM 18Kb instances, each can be configured to 4Kb x 4, 8Kb x 2 or 16Kb x 1. BRAMs can have dual ports for the same instances, allowing performance of half the latency.

4.2.2.2.2 Distributed ROM

Xilinx FPGAs offer another type of memory, which is LUT, distributed ROM, which can be configured to hold design parameters. Each LUT can be configured as 6 input 1-bit ROM (64x1). Normally, distributed ROMs grant more speed than BRAMs, which makes them a good approach to ZynqNet implementation.

4.2.2.3 Initializing ROM

ROMs are usually initialized with Verilog system task $readmemb/ $readmemh inside initial procedural block, which loads memory contents from a file, specifying start and end addresses.

4.2.2.4 ROM Design

For the layer implementation mentioned, each MAC block is responsible for a one filter fetching, this eliminates the option of fitting all design parameters into a huge ROM block, for this will cause high latency. For example, a 10K memory mapped to 64 MACs would require 64 cycles to fetch a new word to convolve, resulting in a low speed convolution process.

4.2.2.4.1 Introducing 3D ROM Array approach

As a result of the previous issue, the approached implementation is to design a separate ROM for each filter in the design. Each ROM is connected to one MAC block, which grants low latency and consistency.

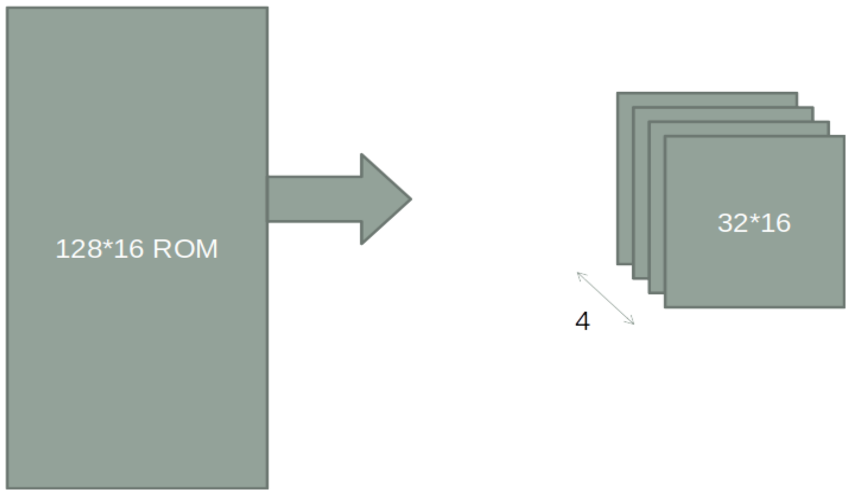


Figure 4-4 3D ROM Array

This implementation yields in N more entries of ROM, where N is the number of ROM subsets, equal to those MACs in a single layer. However, this implementation is beyond BRAM capability, since ZynqNet architecture has 4448 filters, and targeted device can utilize only a lot less than 2940 BRAMs, whilst LUTs are consumed in a huge workload, so a hybrid implementation of ROM is considered as shown in Figure 4-4. Xilinx Vivado offer support for 3D BRAM implementation, whilst 3D ROM is not supported. This leads to a point in which a software approach is required to generate the required ROMs.

4.2.2.4.2 3D Kernels ROM Implementation

The approach here is to generate as many ROMs as number of filters per layer, so a bash script is used for this operation, due to its power in files and I/O handling. It's responsible for the following:

- Divide the layer memory into a number of files that are mapped to each filter
- Generate a ready-for-synthesis SystemVerilog module that contains the array of 2D ROMs

Using this approach, every MAC can have a new input every clock edge, since a separate ROM is responsible for fetching a single word out of each filter, resulting in a latency of 1 cycle per fetch. The previous implementation works perfectly when the filter volume is a power of 2, because other than that, there will be some overheads which can be calculated with the following equation

$$\text{Overhead} = 2^{*}\text{Ceil}(\log2(\text{Depth}))-\text{Depth}$$

Which results in 0 overhead in case of a power of 2 depth. For example, to implement 1x1x576x32 Layer kernels, that would be 32 blocks of 1024 depth each; since 576 needs to be addressed with 10-bit vector.

To get rid of these overheads, the solution is to invoke a second bash script, which is responsible for the following:

- Divide each memory subset into another two subsets, one of them is a power of 2
- If the other part is a power of 2 then script ends
- If the other part is still not a power of 2, it is guided into the second script again

Output is the ready-for-synthesis System Verilog 3D ROM module

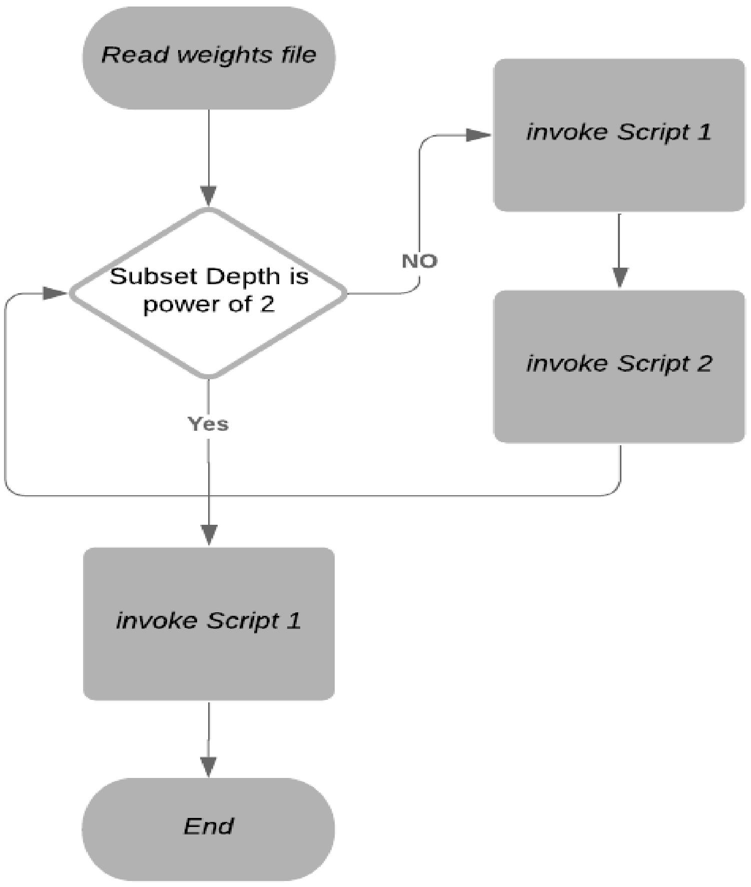


Figure 4-5 General Steps of ROM generation

This approach shown in Figure 4-5 results in 0 overhead in case of power of 2 depth. For example, to implement 1x1x576x32 Layer kernels, that would be 32 blocks of 1024 depth each; since 576 needs to be addressed with 10-bit vector.

Figure 4-6 shows an example of the ROM generation for 32 filters, with 576 words:

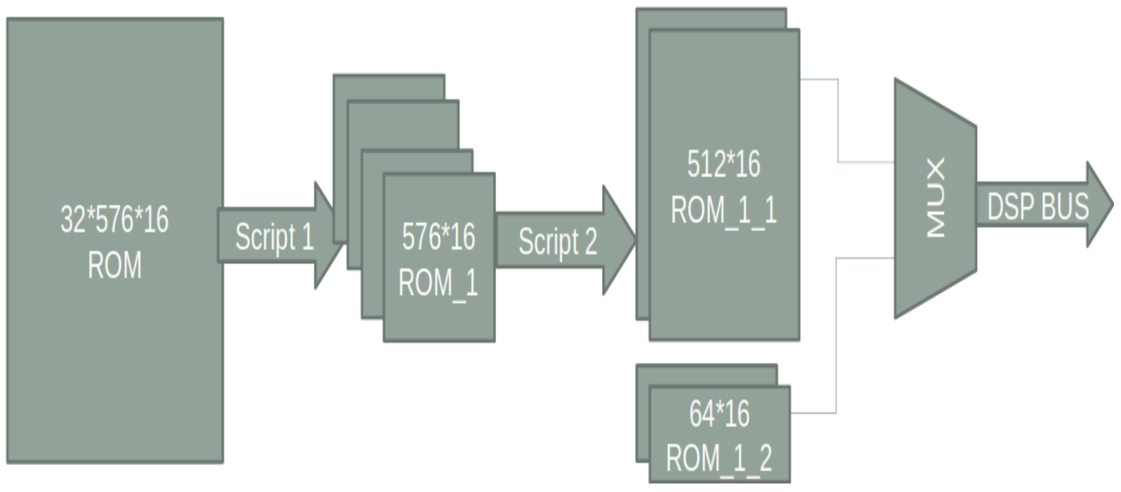


Figure 4-6 ROM generation for 32 filters with 576 words

4.2.2.5 Bias ROM Implementation

Bias parameters are 32-bit fixed parameters that are added to the feature maps before moving to ReLU. They can easily fit inside a System Verilog initialized registers, however, they are generated using a specific script as well to save time, effort and avoid errors.

4.2.3 Intermediate Storage

4.2.3.1 *Shared Memory Using 3D BRAMs*

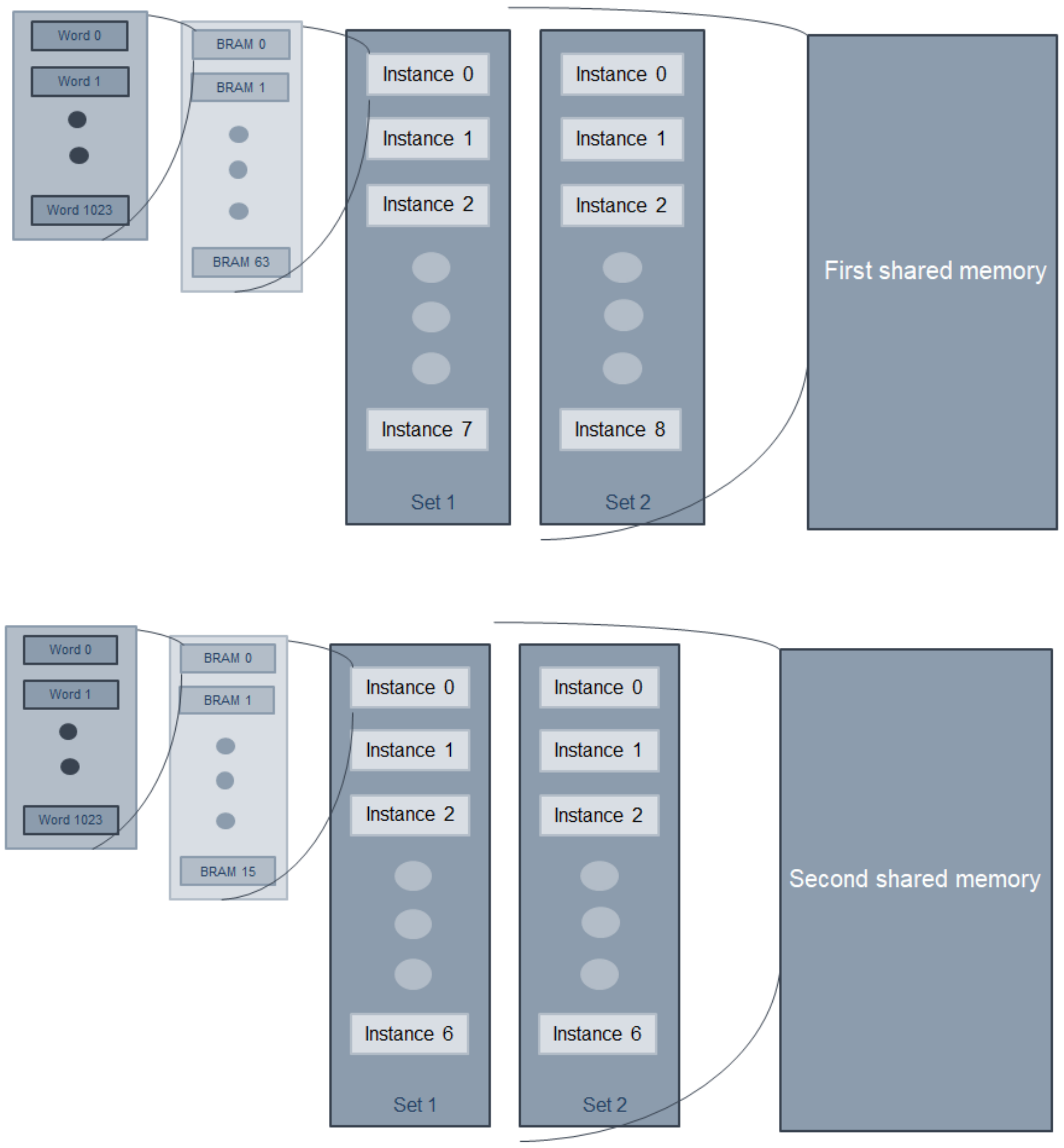


Figure 4-7 Used implementation for feature maps storage

Each layer has output feature maps that have to be stored in order to be passed to the next layer, storing can be done in memory or cache using different resources in FPGA. The number of words of output feature maps of Conv1 layer is 128x128x64=1048576 words, so 16,777,216 flops are needed in order to store these feature maps in a cache using flip flops which is a massive number for storing only one layer so this implementation can't be used. But if the output feature maps of

Conv1 layer get stored in memories using BRAMs, 1024 BRAMs will be needed which is considered acceptable utilization of resources, but by calculating the total number of feature maps that need to be stored, a very huge number of BRAMs will be needed, in addition to that, BRAMs has a queuing problem as it's needed to store multiple feature maps in the same clock cycle while BRAMs can't store more than 2 inputs in one clock cycle, so this implementation can't be used either.

Used implementation shown in Figure 4-7 is a modification of the previous implementation using BRAMs to avoid the problems caused, a shared memory is used for all layers so that they can store and write from the same memory, but to avoid conflict between storing and reading in the same memory, two shared memories are used and each layer will be either reading from it or storing in it, and the next layer will be the opposite from what the previous layer done e.g. if Squeeze2 layer read from shared memory #1 and stored its feature maps in shared memory #2, then Expand2 layer will read from shared memory #2 and will store in shared memory #1.

Another modification to solve the queuing problem is using 3D BRAMs (array of BRAMs), and as feature maps consists of multiple channels so elements of each channel can be stored independently in the same clock cycle in different BRAM. So, this modification solved the queuing problem and allowed the BRAMs to be used as storage for feature maps between layers.

4.2.3.2 Storing Intermediate Storage

The size of the shared memory will depend on the maximum number of output feature maps from any layer which is Conv1 layer for shared memory #1that needs 1024 BRAMs, and Squeeze8 layer for shared memory #2 which needs 112 BRAMs. Each BRAM in the array will contain elements of one channel only, but the elements of that channel can be stored in multiple BRAMs as one BRAM only may not be enough to store all the elements of the channel.

Another modification is needed as some layers requires padding to the feature maps of the previous layer to output specific dimension so this problem can be solved by storing the padded zeros while storing the feature maps of the previous layer e.g., Squeeze2 has padding of 1 so the feature maps of Conv1 should be padded then while storing these feature maps, zeros will be added in their supposed locations and Squeeze2 will read the padded feature maps instantly.

This modification requires increasing the number of BRAMs needed so shared memory #1 will need 1088 BRAMs while shared memory #2 needs to be duplicated because this shared memory will always be read by 2 parallel layers in the same time (Expand Layers) and one of them needs padding while the other doesn't need it then output feature maps of the previous layers will be stored in 2 memories at the same time but one of them will be handled to pad the output feature maps, the final number of BRAMs needed by shared memory #2 is 224 BRAMs.

4.2.3.3 Reading from Intermediate Storage

Regarding reading the feature maps from BRAMs in order to pass them to the next layer, an algorithm is needed to get the windows in order, taking into consideration the stride of each layer as shown in Figure 4-8. If the filter size is 3*3, taking the element of each window in the right order means it is needed to take three elements of each row of three consecutive rows using two parameters {row, col}, row changes from 0 to 2 and col changes from 0 to 2 for each row. Then if there is a stride the window will be shifted with the value of stride.

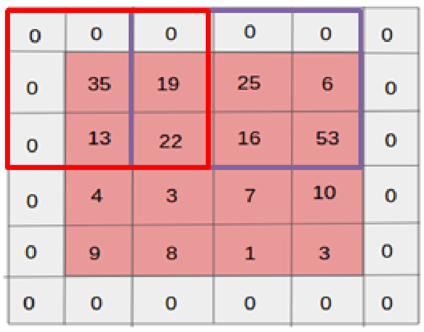


Figure 4-8 Convolution operation with Padding =1, Stride=2

As mentioned before, feature maps of each channel can be divided into multiple BRAMs so for example the first 1024 outputs of the first channel are stored in the first BRAM in the first instance then the second 1024 outputs are stored in the first BRAM in the second instance and so on, so the address of the element that will be read from the BRAMs needs to be zero when it reaches 1024 or multiples of 1024. That can be done by subtracting the needed address by multiples of 1024 in order to read from the right instance, for example if it's needed to read an element from instance 8 then address = the actual address -8*1024.

4.2.3.4 Controlling Operation of Layers

In order to control the operation of the layers in the desired sequence, the shared memories control the sequence as follow; first, the start signal is sent from the top module to the first layer Conv1 to run it and at the same time it is sent to shared memory #1 module in order to prepare the storing signals and parameters. When Conv1 finishes operation, it sends the end signal to shared memory #1 module to prepare the reading algorithm and parameters of the second layer Squeeze2.

When shared memory #1 module gets ready, it sends the start signal to Squeeze2 and at the same time it is sent to shared memory #2 module in order to prepare the storing signals and parameters and so on until the last layer Conv10 as shown in Figure 4-9.

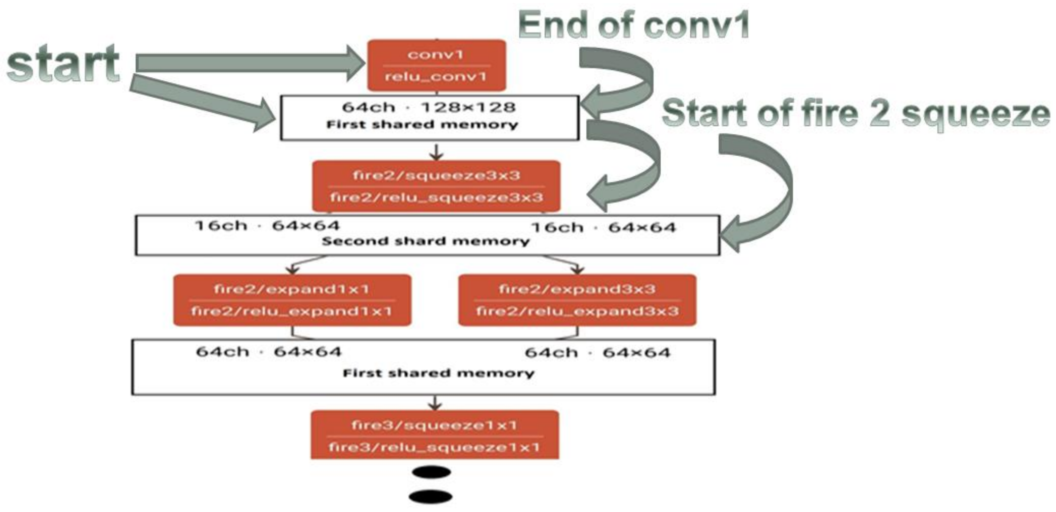


Figure 4-9 Control through memory modules

4.3 Average Pooling layer implementation

Usually in CNN algorithm, convolutional layers are interleaved with pooling layers to reduce the dimensions of its input feature maps retaining the most dominant information. These layers may be average or maximum pooling layers. In ZynqNet, Global average pooling is used to reduce the input feature map from 8*8 with 1024 input channel to 1*1 with 1024 input channel as shown in Figure 4-9.

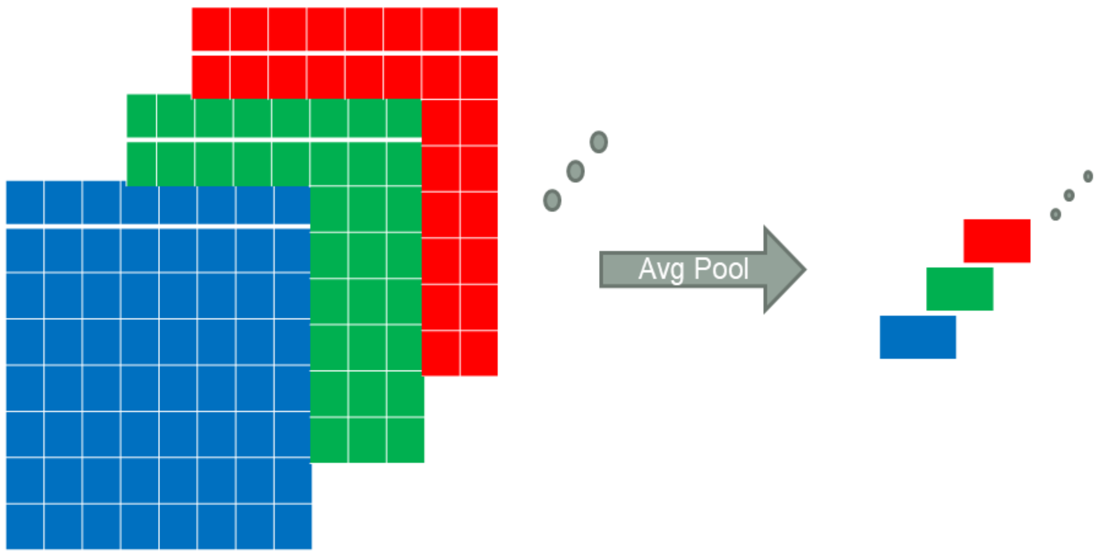


Figure 4-9 Average Pooling Operation

4.3.1 First approach of implementation

To implement average pooling, it is required to get the sum of all elements in each channel divided by number of elements. As a first approach to implement this layer is pipelining the output feature maps Conv10 stored in registers with the usage of adder tree for each channel to get the final output as shown in Figure 4-10.

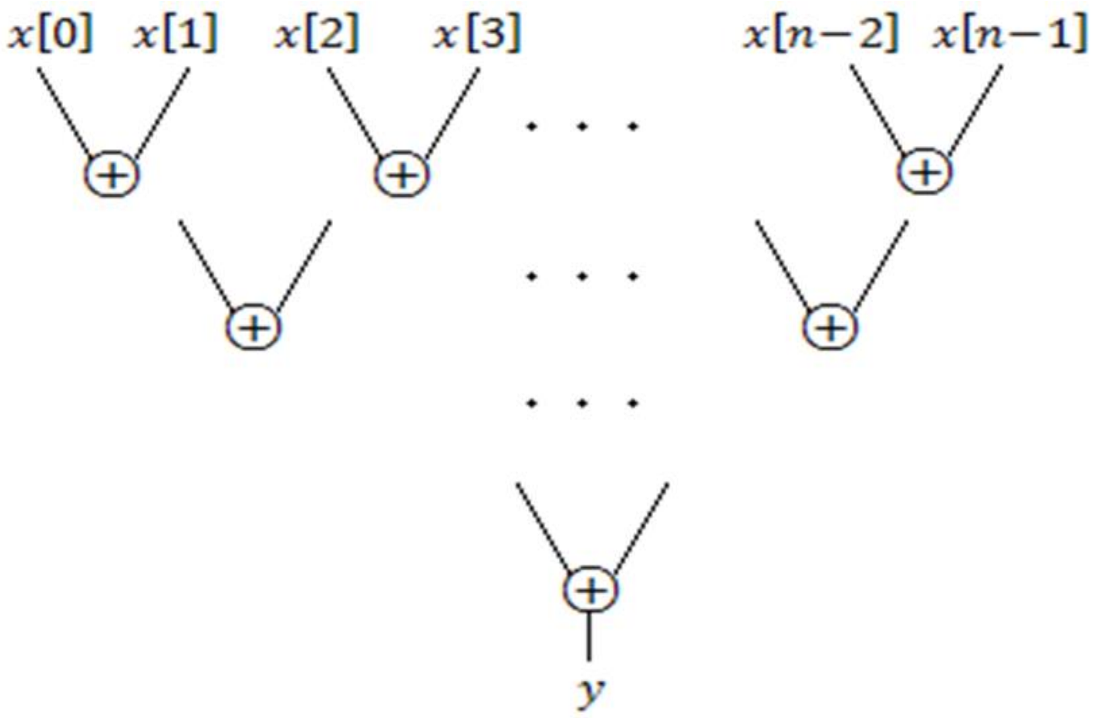


Figure 4-10 Implementation of Average pooling using Adder Tree

In ZynqNet, Conv10 contains 1024 output channels with 8×8 elements each so the total number of needed registers is 65,536. The total number of stages is dedicated by $\log_2 k$ where k is the number of input feature maps which equal 64 and the number of adders in one channel is dedicated by $\sum_{n=0}^{n=(\log_2 k)-1} 2^n$ so average pooling can be implemented with 6 stages pipelined and 63 adders in one channel. The total number of adders in all channels is determined by multiplying the number of adders in one channel by the number of instances used, a shift unit to be used as an approximation of division for averaging because the number of pixels is power of 2 as shown in Figure 4-11. This implementation requires 1024 cycles to finish if only 1 instance is used. To improve timing and throughput 8 instants of average pool is used in parallel to reduce the time to 128 cycles. Hardware implementation of this approach uses 8k LUTs and 8k FFs.

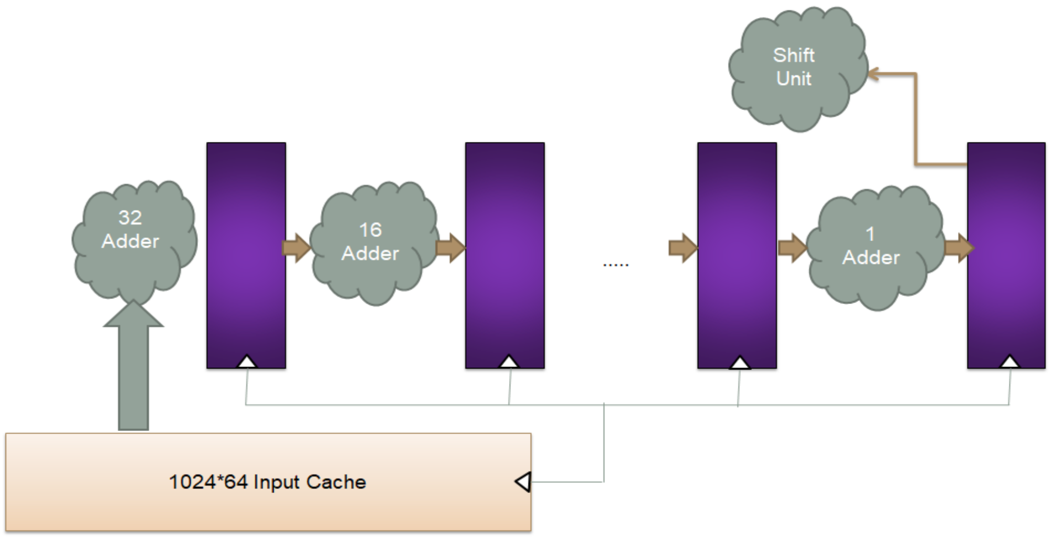


Figure 4-11 First approach of average pooling using 6 stages pipeline

4.3.2 Second approach of implementation

As the number of needed registers to store the output feature maps of Conv10 is very huge, a second approach is introduced which is mainly based on not storing the output feature maps of conv10 but using pipeline and inputting them directly into an array of accumulators as shown Figure 4-12.

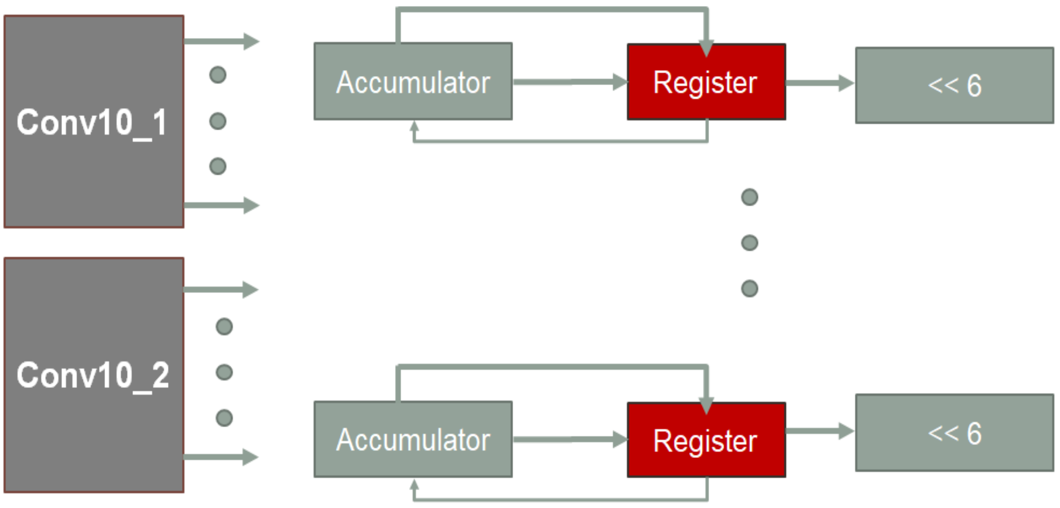


Figure 4-12 Second approach of average pooling using full pipeline

After 64 cycles of accumulating, the sum of all elements in each channel will be ready in the registers. Then they will be divided by 64 using the shift unit (shift right by 6). The number of needed accumulators is equal to the total output channels from Conv10 which is 1024 so it consumes around 33,000 LUTs and around 33,000 FFs.

4.3.3 Third approach of implementation

In order to reduce the previous utilization, the design approach is turned to the third and final approach which is mainly based on that the two splitted parts of conv10 that don't run simultaneously in this design so a good designer can use only half of this numbers of accumulators for two consecutive times once for Conv10/split1 and another for Conv10/split2 as shown in Figure 4-13. In this approach a multiplexer is needed at the input of the accumulators array to select between the output of Conv10/split1 and the output of Conv10/split2. This approach consumes around 10,000 LUTs and around 18,000 FFs.

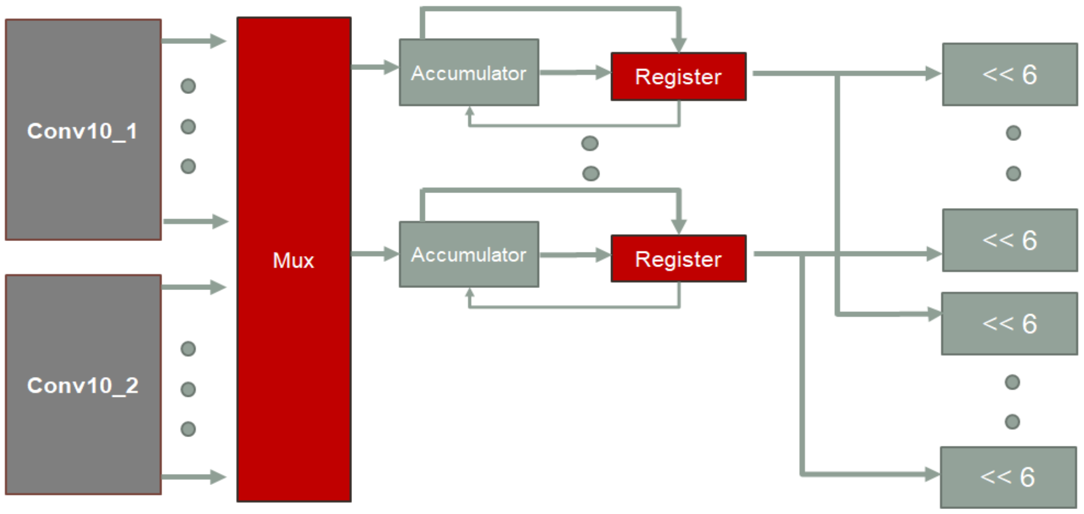


Figure 4-13 Third approach using half pipeline

4.4 Output prediction implementation

It is required to compare the 1024 outputs from average pooling to get the maximum number which is an index that refers to the output prediction. One comparator and two registers are needed as shown in Figure 4-14. First output is compared with the value of the register which is initially equal to zero. Then the value of greater element is saved in the value register and the index of the greater one is saved in the index register. Second output compared with the value register and the value and the index of the greater element is saved and so on until the 1024 outputs are compared. At this time, the index register contains to the index of the output prediction.

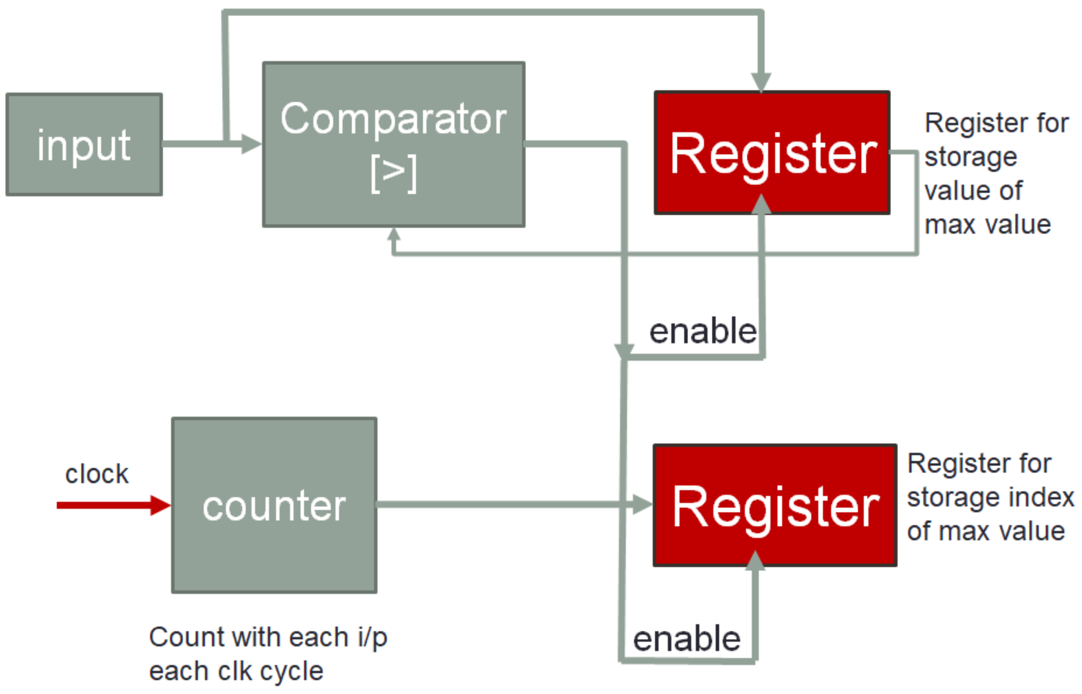

input
Comparator [>]
Register
Register for storage value of max value
enable
clock
counter
Register
Register for storage index of max value
Count with each i/p each clk cycle
enable

Figure 4-14 Output prediction implementation

4.5 Summary

This chapter provides a discussion on the project’s chosen design with the details of how the convolution layers of ZynqNet are implemented, and introducing the different approaches of implementing kernel storage and input feature map storage, then the proper implementation of pooling layer, to end up with output prediction implementation for getting the index of the desired output.

Chapter 5: Optimizations and Verification

5.1 Optimizations for ZynqNet implementation

RTL design, especially large designs, is vulnerable to waste of resources, inefficient modeling techniques and unaware-of-power structures that a designer should keep in mind when achieving an optimal design. The very first stage of the proposed design is to meet functionality at any price, then paving the way into different optimization techniques, of which some trade-offs are sacrificed and maybe eradicating totally unused blocks or components. A good designer should compromise what he is trading off in his optimizations; hence, the following aspects were categorized to make the most out of the required optimizations:

- Area aware optimizations
- Timing and Pipelining
- Power consumption reduction
- Placement Optimizations
- Routing enhancements

5.1.1 Area Aware Optimizations

ZynqNet CNN architecture is quite large, compared to its predecessor SqueezeNet, so optimizations in terms of area are urgently needed to reduce power consumption, PAR issues and improve circuit clocking. Following are the most effective area optimizations achieved:

5.1.1.1 Control Sets Reduction

A control set is the grouping of control signals (set/reset, clock enable and clock) that drives any given SRL, LUTRAM, or register. For any unique combination of control signals, a unique control set is formed. The reason this is an important concept is that registers within a 7 series slice all share common control signals and thus only registers with a common control set may be packed into the same slice. Designs with several unique control sets may have many wasted resources as well as fewer options for placement, resulting in higher power and lower performance. Designs with fewer control sets have more options and flexibility in terms of placement, generally resulting in improved results. Optimization in control sets is done on two phases:

- Removing enables/resets/presets on registers that may not be affected by initial states
- Using control_set_merge switch in opt_design post synthesis

This type of optimization allows the placer to freely place cells all over the fabric, without wasting slices on some unique cells. However, this optimization frees Area in account of power, as the switching activity on clocked cells increases, therefore consuming more dynamic power.

5.1.1.2 Re-quantization of the fully connected layer

Fully connected layer (Conv10) holds the greatest number of parameters in ZynqNet CNN, using most of the memory resources allocated to the device. Hence, re-quantization of this layer from 16 to 15 bits should make a slight difference in area, without harming accuracy; based on simulation results. Re-quantization of Conv10 to 10 bits is a must in order for the design to operate freely at higher frequencies.

5.1.1.3 Packing of expand parameters

Fire modules in ZynqNet consist of a Squeeze and 2 Expand layers. Usually, the Expand layers are working simultaneously, so instead of allocating separate memory for each, packing both Expand parameters in one memory reduces the area needed for fetching. This however harms the comfort ability of the placer to freely place memory cells adjacent to DSPs, but the tradeoff for area is so much better.

5.1.1.4 Registering high fan-out nets

Xilinx Vivado recommends fan-out of no more than 10000 for nets to operate correctly, however, in the presence of tight timing constraints and high area usage, nets should not fan-out to any more than 500 cells, which means almost 20x the area. Proposed design has all high fan-out nets registered, so that replicated cells consume flops and not the precious fabric LUTs. Xilinx Virtex-7 allows 8 flops and 4 LUTs per slice, so replicating registers would not be an issue. This, however, contradicts with the reduced control sets, so the replicated nets were chosen carefully whether they have unique control sets or not.

5.2 Timing and Pipelining optimization

Initial design for ZynqNet has hardly met 100 MHz constraints on routed design in OOC mode, so pipelining had nearly been a must for the design to operate freely at 100 MHz, The pipelining phase was all about breaking critical paths, which by design nature is applicable to many nets, given the high fan-out of the design nets.

Paths broken by pipeline:

- Paths with logic levels more than 5
- Paths that have DSPs input pins as end points
- Paths that have BRAMs output pins as start points
- Paths suggested by Vivado pipeline analysis

5.3 Power consumption optimization

5.3.1 Pushing pipeline registers into Big Blocks

DSPs as in Figure 5-1 and BRAMs have optional Input/output pipelining registers, which are meant for timing and power optimization. Pushing the pipeline flops or even non-pipeline flops into Big Blocks improve timing paths, however, this is a compromise between area, timing and power versus placement comfort ability, as pushed registers could have been better-placed prior placement with physical optimization provided by Vivado, given that proposed design is high area consuming.

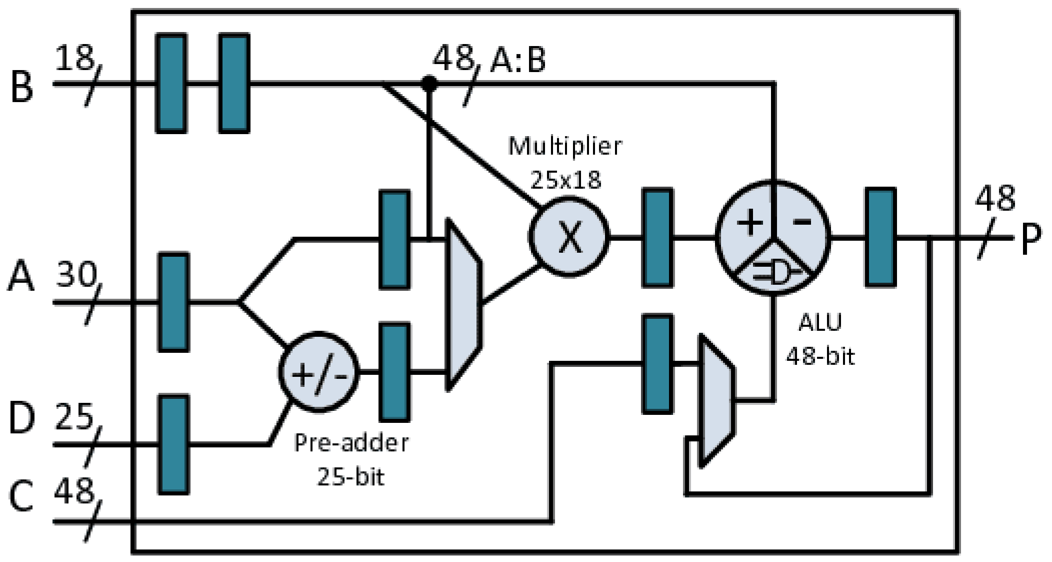


Figure 5-1 DSP

5.3.2 Clock Gating on Enable Pins

Proposed design is FPGA based, hence, use of clock gating is limited to **BUFCE** clock buffers, but the same results can be approached by moving the clock gating cells to CE pins of flops. Clock Enable pins offer the same power reduction as clock gating techniques, with the same level of safety assured. This however requires more unique control sets, contradicting the rule of thumb provided by Xilinx (Not to consume more than 7.5% of control sets); hence, the choice of clock enabled flops was based upon the number of flops per unique enable signal, vulnerability to replication and average expected switching activity.

5.4 Placement and Routing optimizations

5.4.1 Manual replication of signals

Initial design implementation could not start routing cells due to high congestion of cells in given directions, with more than 140% congestion in South and 126% congestion in East. Routing congestion normally occurs due to high area usage, tight timing constraints and high fan-out nets. Proposed design has been synthesized with **KEEP** hierarchy options on sub-modules, which blinds the synthesizer from max_fanout attribute on synthesis. Hence, high fan-out nets were replicated manually, surely on account of increased area. Manual replication of high fan-out nets allows the placer to freely replace critical cells and rearrange block placement according to new replicas sites.

Result from such process can be seen clearly in the wiring and placement congestion in Figures 5-2 and 5-3.

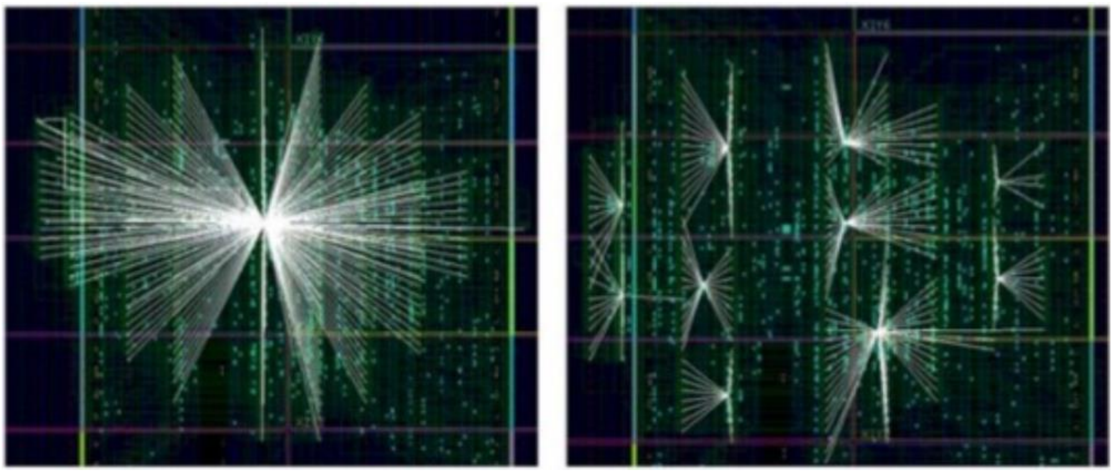
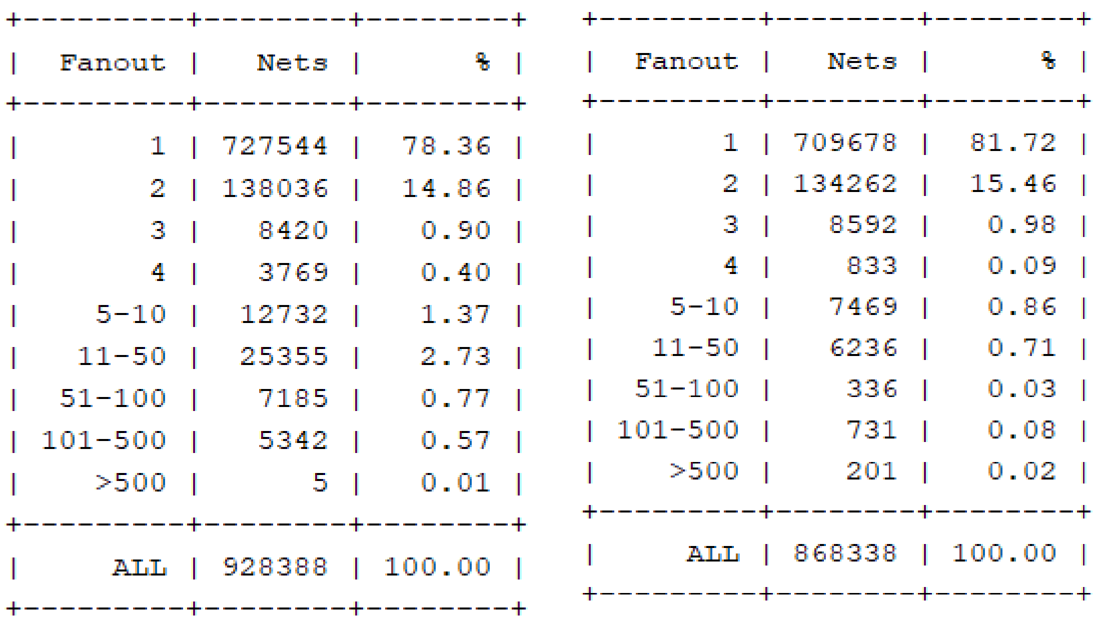


Figure 5-2 Wiring congestion



```
+----------+--------+--------+
|  Fanout  |   Nets |      % |
+----------+--------+--------+
|        1 | 727544 |  78.36 |
|        2 | 138036 |  14.86 |
|        3 |   8420 |   0.90 |
|        4 |   3769 |   0.40 |
|     5-10 |  12732 |   1.37 |
|    11-50 |  25355 |   2.73 |
|   51-100 |   7185 |   0.77 |
|  101-500 |   5342 |   0.57 |
|     >500 |      5 |   0.01 |
+----------+--------+--------+
|      ALL | 928388 | 100.00 |
+----------+--------+--------+
```

```
+----------+--------+--------+
|  Fanout  |   Nets |      % |
+----------+--------+--------+
|        1 | 709678 |  81.72 |
|        2 | 134262 |  15.46 |
|        3 |   8592 |   0.98 |
|        4 |    833 |   0.09 |
|     5-10 |   7469 |   0.86 |
|    11-50 |   6236 |   0.71 |
|   51-100 |    336 |   0.03 |
|  101-500 |    731 |   0.08 |
|     >500 |    201 |   0.02 |
+----------+--------+--------+
|      ALL | 868338 | 100.00 |
+----------+--------+--------+
```

Figure 5-3 Placement congestion

5.4.2 Removing Global Reset Signal

By default, reset signals are the second highest fan-out signals, with clock signals being in the first place, so in bulky designs, resets should be handled carefully. Unlike clocks, reset signals have no dedicated buffers and global routing, such as **BUFG** or **BUFR**, meaning the mostly depend on local routing resources.

Many FPGA designers ignore the fact that FPGA fabric comes with a global initial state parameter that may help them in their design, in fact, FPGAs have embedded global reset that reprograms the chip into the initial state or factory reset. Dependence on such utility, routing has been totally improved whilst keeping the same function.

5.5 Verification of RTL functionality

Continuing in the FPGA design flow, the point of behavioral simulation is reached to make sure the functionality of the design is met with the required specifications. Taking into consideration that every optimization step taken (as explained above) would lead to re-verifying the code again to make sure that the optimization didn't affect the functionality.

The functionality of this CNN architecture is to predict the class of the input image, so multiple images from ImageNet were tested and compared to the results predicted from Python.

5.5.1 Testing Strategy

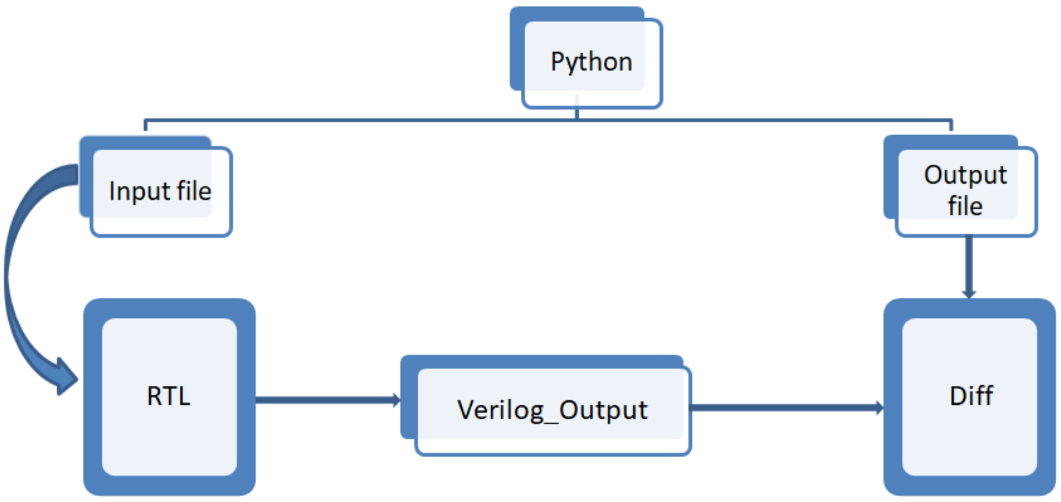


Figure 5-4 Testing Strategy

As ZynqNet architecture was first implemented on Python, so it will be the reference for the hardware implementation to check the sanity of it. Each layer was tested on its own to guarantee it functions right.

As in Figure 5-4, the input for each layer was extracted from Python in files with the correct order it's designed to enter the layer with, these files were fed to the test bench created for each layer.

After passing through the flow of the layer, the output of each layer is written in files then by using diff command on Linux, the two files; one from Verilog and the other from Python are compared to see if the output is the same as expected or not. If the output is not the same, the code would be modified and retest it again.

5.5.2 Simulation Results

Figure 5-5 is a sample from ImageNet validation set which is a "schooner "

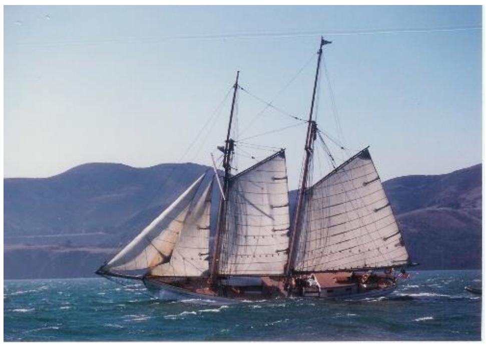
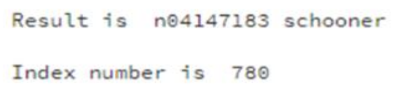


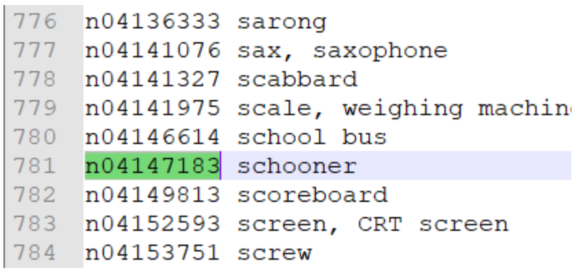
Figure 5-5 Sample Image

Figure 5-6 and 5-7 shows the Python results predicted correctly with index predicted 781 (starting from index 1).



```
Result is  n04147183 schooner

Index number is  780
```



```
776 n04136333 sarong
777 n04141076 sax, saxophone
778 n04141327 scabbard
779 n04141975 scale, weighing machin
780 n04146614 school bus
781 n04147183 schooner
782 n04149813 scoreboard
783 n04152593 screen, CRT screen
784 n04153751 screw
```

Figure 5-6 Python's Output

Figure 5-7 Sysnet_words

Figure 5-8 shows the RTL behavioral simulation results which is similar to Python results with the output predicted correctly, index 780 (starting from index 0) after 80 ms.

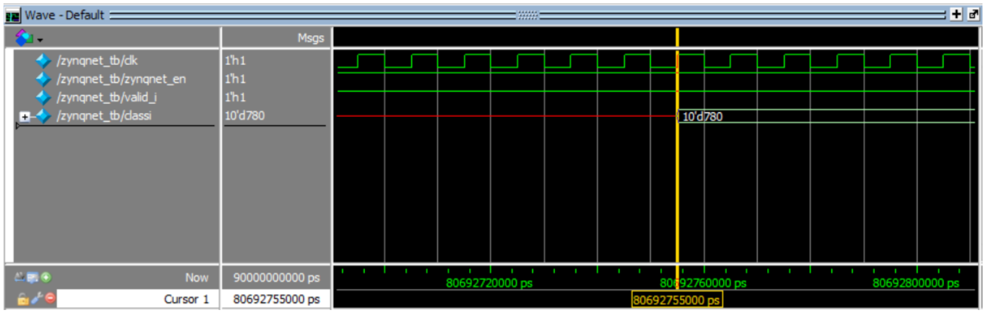


Figure 5-8 Output Waveform

5.6 Summary

This chapter includes different optimization techniques and the effect of them in different fields; it also discusses the verification process done to ensure that the functionality of the design is as specified.

Previous optimization techniques have great tradeoffs, but a good designer should pick up the best strategy and compensate for the sacrificed performance or speed. This, however, is an addition to the great optimization that Xilinx Vivado is capable of, in either of the previous fields. Noting that design accuracy has not change, Table 5-1 shows the optimizations made, pros and cons and tradeoffs:

Table 5-1 Pros and cons of every optimization

	Area	**Power Consumed**	**Timing**	**PAR**
Re-quantization	better	better	-	-
Control sets reduction	better	worse	-	-
ROM re-arrangement	better	Slightly worse	-	worse
Registering High Fanout	better	Slightly worse	better	-
Pipelining	Slightly worse	better	better	-
Big Block I/O Registers	-	better	better	worse
Clock Gating	Slightly worse	better	-	-
Manual Replication	worse	worse	better	better
Removing Reset	-	Slightly better	-	better
Vivado opt_design	Slightly better	Slightly better	-	-

Iterative tests were made after every optimization technique to make sure that the functionality is still valid.

Chapter 6: Synthesis and Implementation

6.1 Synthesis Flow

FPGA synthesis is a vital phase in the deployment of ZynqNet CNN architecture, as it is the anchor of all phases to come, including optimization and PAR. Xilinx Vivado offers a variety of options and switches for synthesis, and even ready-to-use strategies that help designers employ Vivado Synthesizer to achieve a target for their RTL, such as area, timing and so on. However, the proposed design has not been a fan of these strategies, since they are not fully customizable, so, the synthesis approach was to manually consider the design limitations, goals, Vivado abilities and use trial-and-error strategy on best expected approaches.

The very first stage of synthesis flow is to determine how the tool would handle hierarchies, either full flat hierarchy, none flat or rebuilt flattening. Full flat hierarchy synthesis allows the synthesizer to break the hierarchy, perform logical optimization across the whole design, but with high runtime, while none-flattening prevents the synthesizer from performing across-hierarchy optimizations, provided that runtime is much less than rebuilt or full flattening.

Table 6-1 shows the results from different flattening options, keeping the other synthesis switches as is

Table 6-1 Results from different flattening options

	Full	**rebuilt**	**none**
LUTs used	96%	85-88%	77-81%
Slice utilization	> 100%	> 100%	96%
Placement-status	over-utilized(fails)	over-utilized(fails)	Ready for placement

Clearly the none-flattening switch was the best choice, even if it had some issues with the high fan-out optimizations, but these issues could be compensated later in the implementation stage.

Finding the best approach was the challenge here, so different combinations of synthesis switches have been used, to conclude in the best custom strategy that would fit the proposed design, however, it was clear that main issues were area and congestion, due to high utilization. Following are the best synthesis directives and options that led to the best results:

- Directive AreaOptimized_high
- None flattening
- Resource sharing on
- Retiming
- Incremental synthesis

6.2 Constraints

Constraints are the user guideline to synthesis tools, giving the user the ability to control almost every aspect in design. Proposed design has been augmented with timing and physical constraints, some of which are meant for synthesis and others are critical in PAR.

6.2.1 Timing Constraints

Proposed design has timing specifications that made constraints much easier; registering every sub-module inputs and outputs, not having multiple clocks and neither false path. Timing analysis for FPGA-based designs have been much easier than ASIC-based designs, because Xilinx Vivado timing engine is so powerful and has great timing models for each cell in targeted device Virtex-7.

6.2.1.1 Clocking

Clocking network in this design is fed by the differential system clock (200 MHz) already soldered into Virtex-7 FPGA board VC709 as in Figure 6-1, connected to a Mixed-Mode Clock Manager cell, that controls the design clock as required, usually determined by STA.

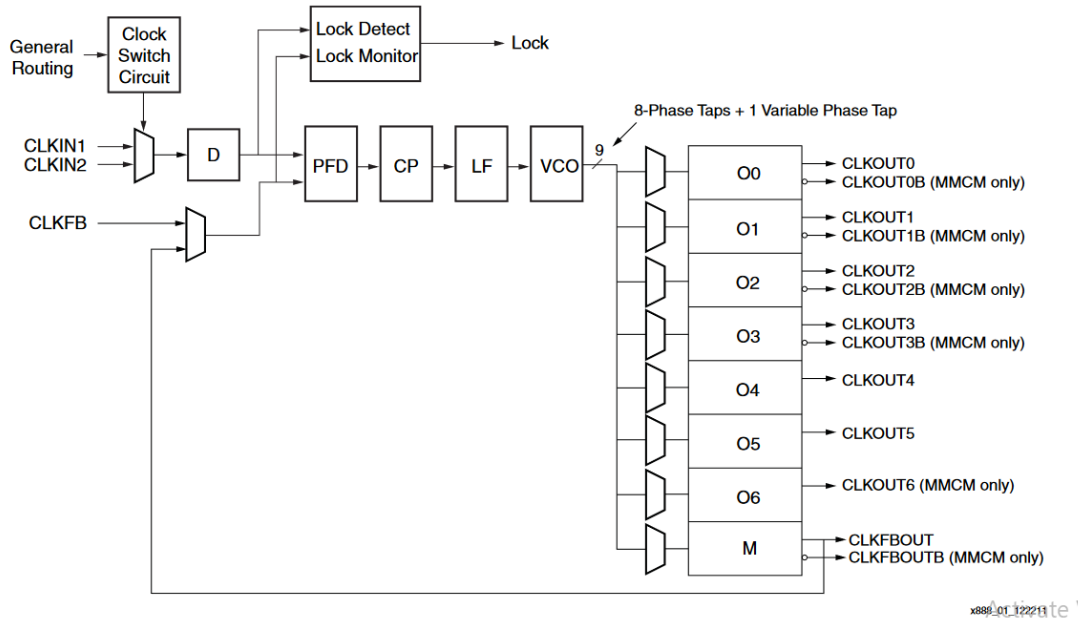


Figure 6-1 Detailed MMCM block diagram

MMCMs (Mixed-Mode Clock Manager) are FPGA primitives that are used for clocking and skew management, they can feed up to 6-7 output clocks, with different frequencies, phases, jitter and feedback control. ZynqNet CNN architecture has been supplied with only 1 **MMCM** primitive, with a total of 4 **BUFG** primitives for clock distribution across the fabric and skew modulation.

6.2.1.2 Constraints

The only needed timing constraint in the proposed design was to **create_generated_clock** from the **MMCM** output port, however easy it is, Vivado made it easier for designers by only using **create_clock** for the input positive system clock, which gives Vivado the ability to optimize clock network, choose best site for **BUFG** and replicate **BUFG** accordingly.

6.3 Physical Constraints

Every design has its unique status and goals, especially in physical conditions such as driving strengths, placement directives, hierarchy control and more. Physical constraints play the role of not giving the permission to synthesis/implementation tool to fully automate the design flow, giving the designer much more power in controlling what is actually going on down there in optimizations and synthesis. Xilinx Vivado offers a variety of physical constraints that almost can cover any user requirements, however, the proposed design invoked a few of these, to control what it was seen to be best not left for Vivado.

6.3.1 RTL Based Constraints

- DONT_TOUCH: On signals and registers that should not be vulnerable to Vivado optimizer efforts, due to restricted physical requirements such as manual replication
- KEEP_HIERARCHY
  - YES: Usually used on instances that lead to less utility when hierarchy-broken
  - NO: In most cases, breaking the hierarchies of big blocks lead to better optimizations
- ROM_STYLE: Choosing whether implemented as Distributed ROM or BRAM inferred
- RAM_STYLE: Mostly used to prevent Vivado from packing registers inside BRAMs
- MAX_FANOUT: Set on nets that have high fan-out if and only if these nets fan-out to pins in the same hierarchy, due to using none flattening option.

6.3.2 Placement Based Constraints

Using Pblocks for floorplanning wouldn't have been the best option for the implementation of the proposed design; hence, floorplanning has been left to Vivado Placer to determine the best placement plan for the netlist. Nevertheless, I/O planning and clock constraining is critical while placement, as they are anchors for the rest of the netlist to be placed.

Placement constraints define the I/O sites, types of I/O buffers used and driving strengths. Target board VC709 has ready-to-use LEDs and push buttons that may be used directly through pin planning. Types of buffers used for module ports are:

- 2 IBUF: for standard input enable signals
- 8 OBUF: for LEDs indicating the classification result
- 1 IBUFDS: Differential to single ended buffer that is used for user clock input buffering

6.4 Implementation Flow

Xilinx Vivado is quite a powerful tool when it comes to timing-aware implementation; however, with the very high utilization of the device, it becomes very hard for the tool to find the optimal solution for netlist implementation and sign-off. High utilization of big blocks (DSPs to around 99% and BRAMs to 97%) made the implementation almost an impossible task; however, with the right procedure and critical selection of processes, Vivado was capable of reaching the final implementation.

6.4.1 Initial PAR

The very first stage of implementation is to optimize design, by clearing the netlist that has just been generated by synthesis. Optimizations chosen are:

- Remap
- Control set opt
- Constant Propagation
- BUFG Insertion

Despite not being so helpful, opt_design is a recommended stage by Vivado for early-detection of non-optimized cells. Due to high utilization, Vivado priorities the correct placement of cells in account of timing driven placement, however, Vivado tries to compromise the effect of utilization when possible. Placement process also includes sorts of optimization such as pin replacement, fan-out optimization and others. Directives chosen in this stage were as follows:

- Quick: for a draft of correct placement without timing or routing consideration, aiming for better runtime.
- Alt_SpreadLogic_high/medium: Those directives help Vivado Router to correctly route the placed netlist

However, the best placement option acknowledged afterwards was the Alt_SpreadLogic_high, due to issues found in later stages. At this point, the netlist was totally un-routable due to high routing congestion, so the following stages were needed.

Next step is Phys_opt; Physical Optimizations by Vivado are set of processes, in which Vivado tries to fix worst negative slack (WNS) and total negative slack (TNS) by doing the following:

- Remap pins according to driving strengths and timing constraints
- Critical cell optimizations such as replication and replacement
- High fan-out and Very High fan-out optimizations on nets
- Pushing register in/out of Big Blocks as to meet timing constraints
- Replacement of clocking resources if possible

This stage is recommended by Vivado if the slack is equal or better than -0.5 ns, and unless WNS is present, this stage would not be performed. Physical Optimizations helps in the overall timing sign-off and fixes routing conflicts, however, the netlist has got better slightly and still was not ready for routing, so the next step is to unplace netlist and replace for a few iterations, using the same directives in the first trial.

6.4.2 Final PAR

The failed implementation process encouraged some area and physical optimizations, resulting in RTL changes that needed to be added to the design checkpoint. This process is called by incremental synthesis, where Vivado synthesizer anchors a design checkpoint, resynthesizes accordingly and produces the new netlist, with better results and reduced run time, up to 90%.

By default, Vivado Placer and Router have iterative property, so the iterations of place_design and route_design have been very useful to correctly implement the design. After a few iterations, Vivado Router could eventually route the design without congestion efforts, with all nets in design. However, post routing timing analysis report was not optimistic at all and resulted in -43 WNS, with routing delay up to 99% of failed paths.

6.4.3 WNS fix and Bit Stream Generation

Routing results were catastrophic timing wise, as the WNS is too high as well as the dramatic TNS. Mainly, the reason behind these results was the nearly-full utilization of device. The solution to such an issue was to iterate on routing sessions, with different efforts and directives to remove negative slacks. The effective procedure was to iterate over the following:

- Route design
    - NoTimingRelaxation
    - MoreGlobalIterations
    - TNS cleanup
- Physical Optimization and high fan-out optimizations

This procedure could eventually fix the timing issues and finally obtain a working floorplan at 100 MHz frequency, ready for bit stream generation and downloading on target board VC709.

6.5 Summary

This chapter provides a discussion on the synthesis and which strategies give the best results, also constraints are handled including physical and timing constraints, then implementation of the design is provided going through placing, routing and fixing WNS then generating the bit stream.

Chapter 7: Results

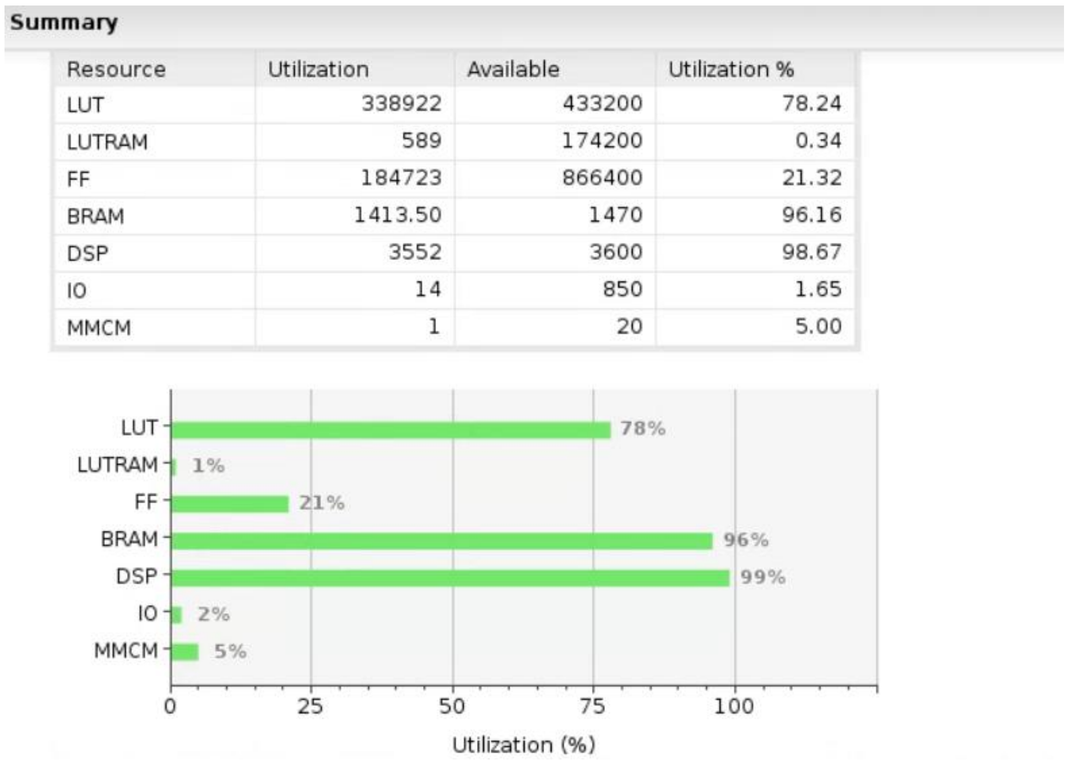
7.1 FPGA Results

The design was first implemented at frequency of 100 MHz then more optimizations were applied until it reached a frequency of 125 MHz, the following will discuss the utilization of the design on the chosen FPGA along with the timing analysis and power consumed at each frequency.

7.1.1 Implementation at Frequency of 100 MHz

7.1.1.1 Utilization

After implementing the design on Virtex 7 VC709 FPGA using the Vivado implementation tool, Figure 7-1 shows the utilization of the resources.



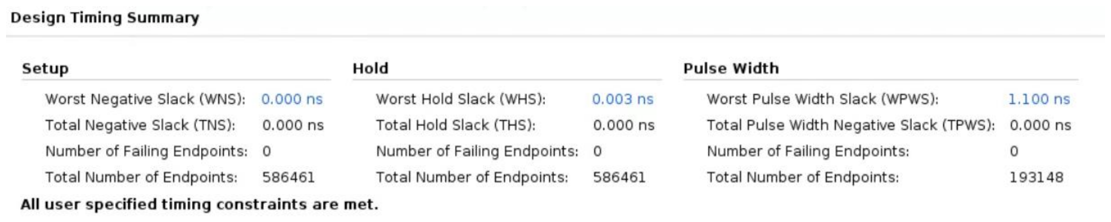
Summary

Resource	Utilization	Available	Utilization %
LUT	338922	433200	78.24
LUTRAM	589	174200	0.34
FF	184723	866400	21.32
BRAM	1413.50	1470	96.16
DSP	3552	3600	98.67
IO	14	850	1.65
MMCM	1	20	5.00

Figure 7-1 Post Implementation Utilization Summary-100MHz

7.1.1.2 Timing Analysis

Figure 7-2 shows the setup and hold time slack, which are equal or bigger than zero which means that the timing constraints are met at clock frequency of 100 MHz



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.000 ns	Worst Hold Slack (WHS):	0.003 ns	Worst Pulse Width Slack (WPWS):	1.100 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	586461	Total Number of Endpoints:	586461	Total Number of Endpoints:	193148

All user specified timing constraints are met.

Figure 7-2 Design Timing Summary -100MHz

Figure 7-3 is a timing histogram at frequency of 100 MHz shows the utilization of the clock frequency used, indicating how good the design is.

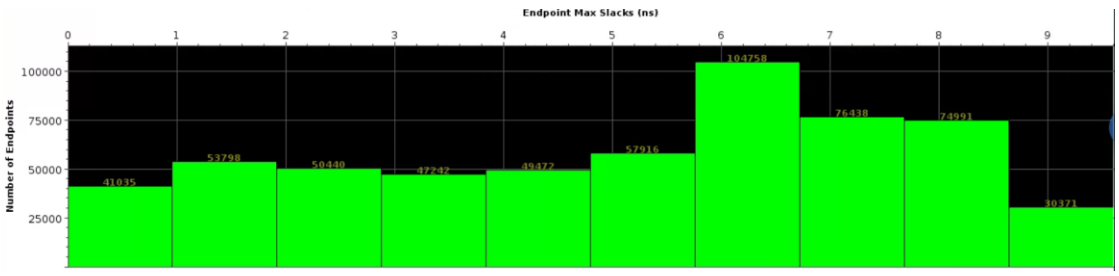
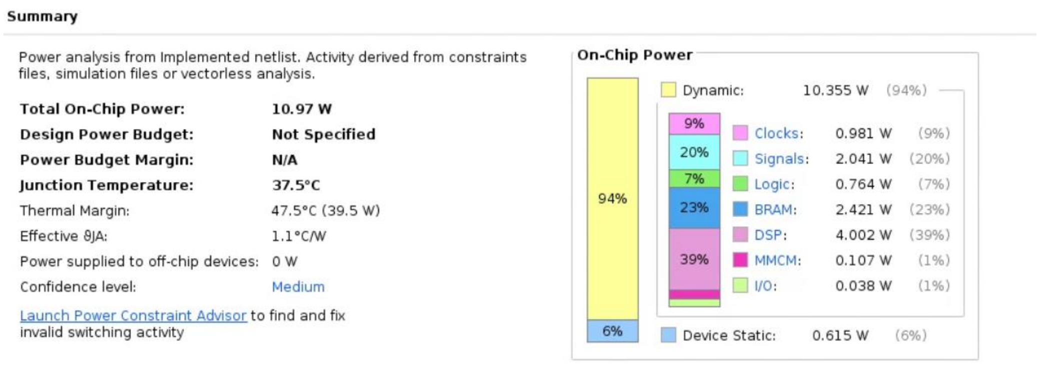


Figure 7-3 Timing Histogram-100MHz

7.1.1.3 Power Analysis

The following Figure 7-4 shows the power consumption of the design on the Virtex 7 FPGA, total power on chip equals to 10.97 watts.



Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**10.97 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**37.5°C**
Thermal Margin:	47.5°C (39.5 W)
Effective ϑJA:	1.1°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:		10.355 W	(94%)
	Clocks:	0.981 W	(9%)
	Signals:	2.041 W	(20%)
	Logic:	0.764 W	(7%)
	BRAM:	2.421 W	(23%)
	DSP:	4.002 W	(39%)
	MMCM:	0.107 W	(1%)
	I/O:	0.038 W	(1%)
Device Static:		0.615 W	(6%)

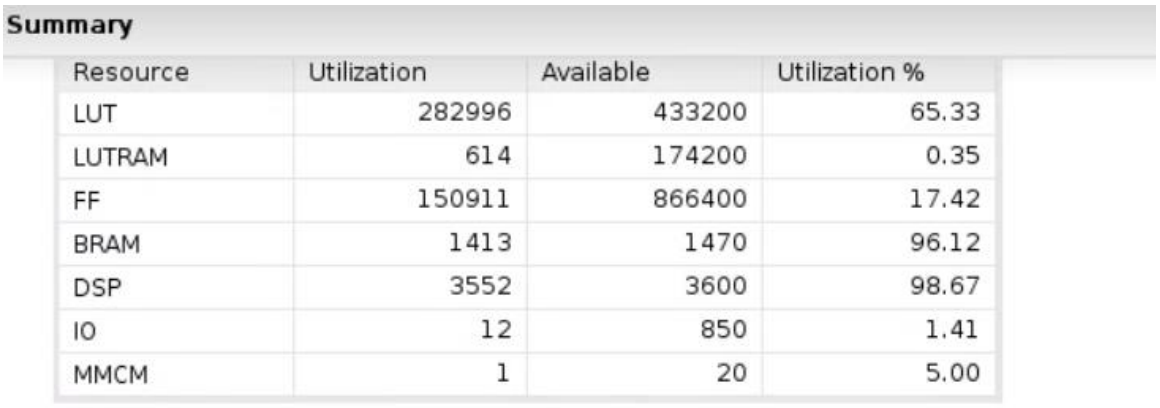
Figure 7-4 Power Report Summary-100MHz

7.1.2 Implementation at Frequency of 125 MHz

7.1.2.1 Utilization

In order for the design to be placed at frequency of 125 MHz, some optimizations had to be made to reduce the used resources and congestion, so fully connected layer weights' were quantized to 10 bits instead of 15 bits taking into consideration that this reduction didn't affect the accuracy.

Figure 7-5 shows the utilization of the resources post implementation.



Summary

Resource	Utilization	Available	Utilization %
LUT	282996	433200	65.33
LUTRAM	614	174200	0.35
FF	150911	866400	17.42
BRAM	1413	1470	96.12
DSP	3552	3600	98.67
IO	12	850	1.41
MMCM	1	20	5.00

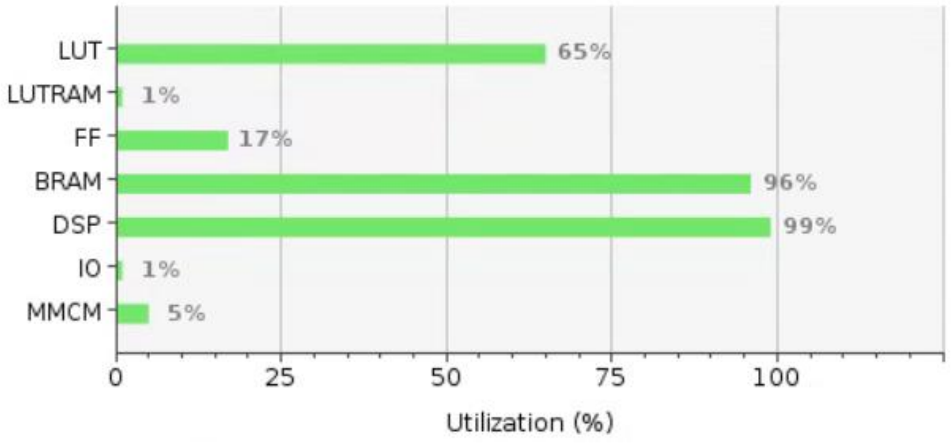
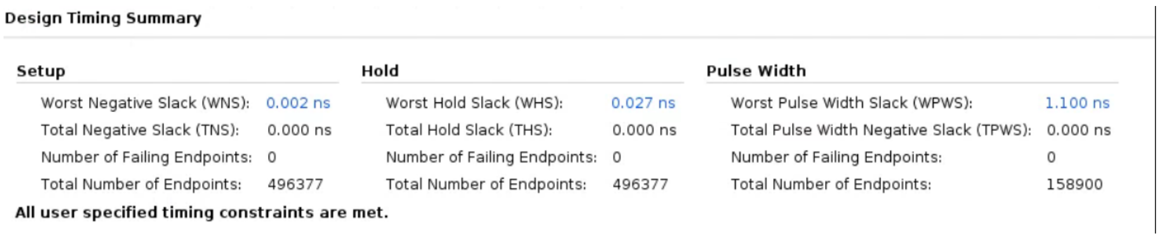


Figure 7-5 Post Implementation Utilization Summary-100MHz

7.1.2.2 Timing Analysis

Figure 7-6 shows the setup and hold time slack, which are equal to zero which means the timing constraints are met at clock frequency of 125 MHz


Design Timing Summary

Setup

Worst Negative Slack (WNS):	0.002 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	496377

Hold

Worst Hold Slack (WHS):	0.027 ns
Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	496377

Pulse Width

Worst Pulse Width Slack (WPWS):	1.100 ns
Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	158900

All user specified timing constraints are met.

Figure 7-6 Design Timing Summary-125MHz

Figure 7-7 is a timing histogram at frequency of 125 MHz shows the utilization of the clock frequency used, indicating how good the design is.

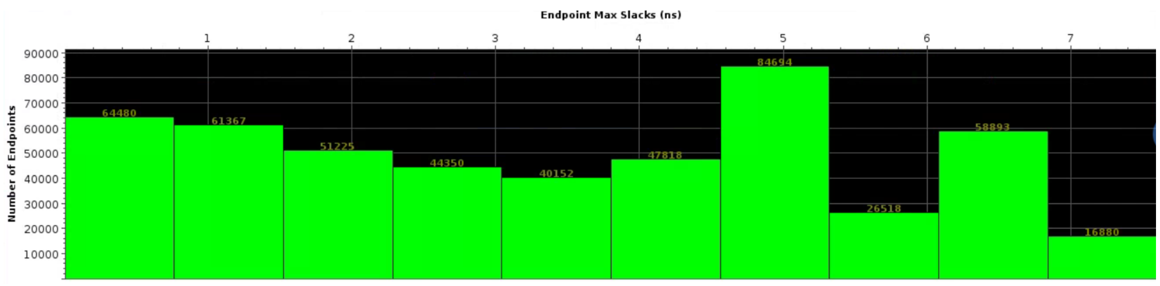
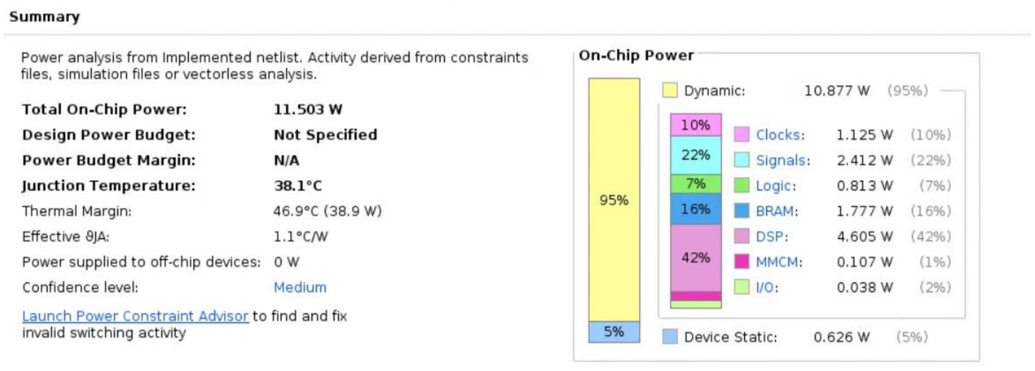


Figure 7-7 Timing Histogram-125MHz

7.1.2.3 Power Analysis

The following Figure 7-8 shows the power consumption of the design on the Virtex 7 FPGA, total power on chip equals to 11.503 watts.


Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**11.503 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**38.1°C**
Thermal Margin:	46.9°C (38.9 W)
Effective θJA:	1.1°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

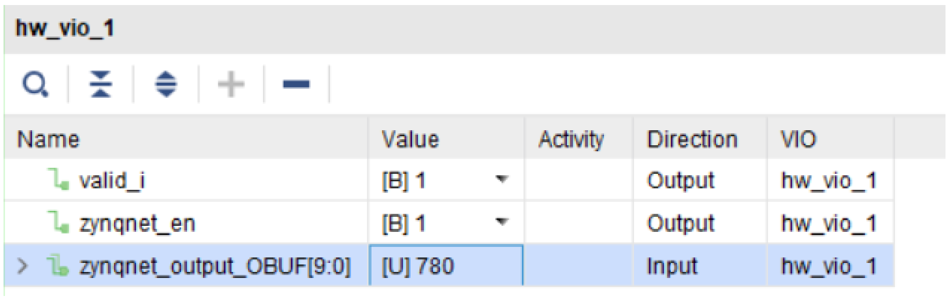
Dynamic:	10.877 W	(95%)	95%
Clocks:	1.125 W	(10%)	10%
Signals:	2.412 W	(22%)	22%
Logic:	0.813 W	(7%)	7%
BRAM:	1.777 W	(16%)	16%
DSP:	4.605 W	(42%)	42%
MMCM:	0.107 W	(1%)	
I/O:	0.038 W	(2%)	
Device Static:	0.626 W	(5%)	5%

Figure 7-8 Power Report Summary-125MHz

7.2 Hardware Testing

The Design was brought up on the targeted FPGA by downloading the bit stream on the VC709 Development kit, the output of the sample image which is the index of the tested image is brought on the LEDs of the kit.

Figure 7-9 shows the input and output probes using VIO, after initiating the enable signals, the output is “780” which indicates the correct index of the predication as discussed above.



hw_vio_1

Name	Value	Activity	Direction	VIO
valid_i	[B] 1		Output	hw_vio_1
zynqnet_en	[B] 1		Output	hw_vio_1
zynqnet_output_OBUF[9:0]	[U] 780		Input	hw_vio_1

Figure 7-9 Hardware probes

Figure 7-10 shows a real life picture of the FPGA after downloading the bit stream; the output in binary is as follows 780 => 11_0000_1100

The first 8 bits are shown in the picture as there are only 8 available LEDs for the output in VC709 Development board.

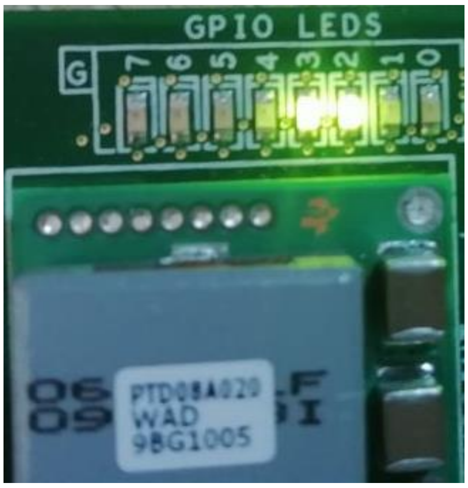


Figure 7-10 Real Life picture for the FPGA

7.3 Summary

Table 7-1 shows the final results from implementing the design with at frequency of 100 MHz and 125 MHz comparing it with the paper in [2].

Table 7-1 Final Results

	ZynqNet paper	Proposed version 1	Proposed version 2
LUTs	154K	339K	283K
DSPs	739	3552	3552
BRAMs(18)	1090	2130	2130
Memory used	On and off-chip	On-chip	On-chip
Fully connected layer quantization	16 bits	15 bits	10 bits
Operating frequency	100 MHz	100 MHz	125 MHz
Power (W)	7.8	10.97	11.5
Inference time (sec)	2	0.08	0.064
Energy/image(J)	15.6	0.88	0.736

Chapter 8: Conclusion and future work

8.1 Conclusion

In this thesis, the implementation and acceleration of CNN on FPGA was discussed by using ZynqNet model which is an optimized CNN in number of parameters and computations. Hardware methodology of implementing the model on FPGA were showed by implementing each layer of ZynqNet separately and using resources of FPGA to store weights of the model on-chip without the need to use external memory, also various techniques and optimizations in the design were demonstrated in order to achieve better results in terms of speed, area and power.

Accelerated ZynqNet was implemented on VC709 Development board and the results proved that using FPGA instead of GPU guarantees higher speed and lower power but also needs much more effort and design time.

8.2 Future work

Future work are more techniques and enhancements that can be applied in the design but are left for the future. These new methods can achieve reduction in utilization of FPGA resources which leads to consuming less power and having higher throughput.

8.2.1 Pipelining inference of multiple images

Currently to make a prediction of the class of a specific image, the image is loaded on the FPGA in the bit stream and the contents of the image is stored on-chip along ZynqNet's kernels, but the current design doesn't support inferring multiple images one by one into the model and having this privilege requires external processor that connects to the FPGA and transfers the images continuously, also pipelining in inferring the images will improve the total throughput, this can be done by adding some control signals which ensures that each layer receives and starts processing next image's data as soon as it finishes processing the previous image's data until output prediction is displayed at the end of inferring of each image.

8.2.2 Using dynamic quantization

ZynqNet's kernels and feature maps in the implemented design are quantized into 16-bits fixed point representation in order to decrease the used memory resources with a slight increase in the error of the model, but quantizing all layer's weights into 16-bits may be a waste of memory resources because some layers can be quantized into less than 16-bits without causing any harm and these wasted bits are considered overhead in the design, so more efficient way of utilizing memory resources is to use dynamic quantization and number of bits will be determined for each layer in ZynqNet to be represented with using fixed point representation, that will also reduce the slight difference between accuracy of the software model and the implemented design on FPGA. Dynamic quantization was used in the fully connected layer (Conv10) by re-quantizing its kernels into 10 bits and generalizing this method on all layers will have a great impact in the design.

References

[1] M. Bojarski et al., "End to End Learning for Self-Driving Cars", 2016.

[2] D. Gschwend, Master Thesis Report “ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network”, 2016

[3] A. Karpathy. “What I learned from competing against a ConvNet on ImageNet”, github.io, 2014, [Online]. Available: http://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet/ [Accessed: 1- Jun- 2020].

[4] A. Khan, A. Zahoora, and A. Qureshi," A Survey of the Recent Architectures of Deep Convolutional Neural Networks", 2019

[5] E. Adel, R. Magdy, S. Mohamed, M. Mamdouh, E. El Mandouh, and H. Mostafa, Bachelor Thesis Report " Accelerating Deep Neural Networks Using FPGA", 2018

[6] "Convolutional neural networks", SlideShare, 2019. [Online]. Available: https://www.slideshare.net/perone/deep-learning-convolutional-neural-networks [Accessed: 10- Oct-2019].

[7] "CNN Architectures: LeNet, AlexNet, VGG, GoogLeNet, ResNet and more…", Medium, 2017. [Online]. Available: https://medium.com/analytics-vidhya/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5 [Accessed: 12-Oct- 2019].

[8] A. Krizhevsky, I. Sutskever and G. Hinton, "ImageNet classification with deep convolutional neural networks", 2012.

[9] "CNN Architectures, a Deep-dive", Towards Data Science, 2019. [Online]. Available: https://towardsdatascience.com/cnn-architectures-a-deep-dive-a99441d18049 [Accessed: 12-Oct- 2019].

[10] F. N. Iandola, M. W. Moskewicz and K. Ashraf, "SqueezeNet: AlexNet-level accuracy with 50xfewer parameters and <1mb model size", 2016.

[11] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network". 2015.

[12] S. Han, H. Mao, and W. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding", 2015.

[13] S. Han, et al. "DSD: Dense-sparse-dense training for deep neural networks.", 2016.

[14] S. Han, et al. "EIE: efficient inference engine on compressed deep neural network." 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). IEEE, 2016.

[15] S. Sombatsiri, et al. "Parallelism-flexible Convolution Core for Sparse Convolutional Neural Networks.", 2018.

[16] A. Deshpande, "A Beginner's Guide To Understanding Convolutional Neural Networks Part 2", adeshpande3.github.io, 2016. [Online]. Available: https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/ [Accessed: 12- Feb- 2020].

[17] "ImageNet Large Scale Visual Recognition Competition (ILSVRC)", Image-net.org, 2017. [Online]. Available: http://www.imagenet.org/challenges/LSVRC/2012/nonpub-downloads. [Accessed: 19- Oct- 2019].