ACCELERATED DEEP NEURAL NETWORKS USING FPGA

By

Esraa Adel Abd El-Sattar

Rana Magdy Ahmed

Sara Mohamed Abdel Wahab

Mona Mamdouh Sayed

Under the Supervision of Associate Prof. Hassan Mostafa

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 Communications Engineering Faculty of Engineering,

Cairo University Giza, Egypt

July 2018

Abstract

Deep Convolutional Neural Networks (CNNs) are the state of the art systems for image classification and scene understating. The target in this field nowadays is its acceleration to be used in real time applications. The solution was using graphics processing units (GPU) but many problems arise due to its high-power consumption which prevents its usage in daily used equipment. The Field Programmable Gate Array (FPGA) became a new solution due to its low power consumption and flexible architecture although the architectures suggested till now have lower speed than GPU due to the limited resources on the kit facing the large number of operations executed in the network. This thesis discusses this problem and providing a solution which compromises between the speed of the network and the power consumption on FPGA.

Keywords: Convolutional Neural Networks; Accelerating CNN; FPGA; Virtex7;

Acknowledgements

We are using this opportunity to express our gratitude to everyone who supported us throughout the graduation project. We are thankful for their aspiring guidance and friendly advice.

First, we want to thank our major advisor Dr. Hassan Mustafa for his encouragement through the whole year, his caring about following up each stage in the project and his suggestions to solve some problems we faced during the project work.

We want to thank Eng. Sherif Maher, FPGA Prototyping Engineer in Mentor Graphics, and Eng. Salma Hassan, Teaching Assistant at Cairo University, for providing their time, experience to help us overcome some obstacles we faced during some stages especially when dealing with new concepts and tools.

Finally, 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.

Table of Contents

List of Tables

List of Figures

List of Acronyms

ALU	Arithmetic logic unit
ANN	Artificial neural network
ASIC	Application-specific integrated circuit
BRAM	Block Random access memory
CLBs	Configurable Logic Blocks
CNN	Convolutional neural networks
Conv	Convolutional layer
CPUs	General purpose processors
DCNs	Deep convolution networks
DNN	Deep neural networks
DRAM	Dynamic Random-access memory
DRR	Double data rate RAM
DSP	Digital signal processing
FC	Fully connected layer
FFs	Flip-flops
FPGA	Field programmable gate array

GPU	Graphics processing units
GEMM	General matrix-matrix multiplication
HDL	Hardware description language
ILSVRC	Large Scale Visual Recognition Challenge.
IOBs	Input/output Blocks
LRN	Local Response normalization
LUT	Look up table
MAC	Multiply and accumulate
MLP	Multi-layer perceptron
MSE	Mean Squared Error
Muxes	Multiplexers
NCD	Native Circuit Description
NLR	No local reuse
NN	Neural networks
OS	Output stationary
PAR	Place & Route
PE	Parallel engine

RAM	Random access memory
RELU	Rectified Linear Unit layers
RF	Register file
RGB	Red-Green-Blue (color model based on additive color primaries)
VHDL	VHSIC (Very High Speed Integrated Circuit) hardware description language

Chapter 1.
Introduction

In this thesis, we are going to propose some techniques to map the deep convolutional neural networks into hardware (FPGAs) in order to be accelerated to fit real time applications which need high speed and less power consumption.

1.1. Motivation

In recent years, artificial intelligence and deep learning have shown their utility and effectiveness in solving many real-world computation-intensive problems. The motivation of these is to eliminate the need of direct programming and create an intelligent system can automatically extract features and recognize a particular pattern and after having learned to recognize a particular pattern, extend that capability to objects that it hasn't actually seen before. In other words, 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. It learns the entire processing pipeline needed to steer an automobile as in [1] by creating models that meet or exceed the ability of a human driver which could save thousands of lives a year.

Among various deep learning algorithms, CNN (convolutional neural networks) is one of the key algorithms for visual content understanding and classification, with significantly higher accuracy than traditional algorithms in many applications, such as image/video processing, face recognition, machine language translation, advances in medicine, autonomous driving and more.

Convolutional neural network (CNN) is first inspired by research in neuroscience, as it's a subset of neural networks (NNs). Neural Network (NN) is a computational model inspired by the way the brain operates, artificial NNs use vast amounts of simple computational elements that are organized in interconnected layers. Modern NNs usually have multiple layers, exceeding 100 in [2] and thus are called deep neural networks (DNNs).

In order to better interpret local features of multi-dimensional inputs such as images, convolutional neural networks (CNNs) are commonly used. CNNs have been shown to be efficient in image-related problems such as classification or scene parsing because the convolution operation captures the 2D nature of images. Also, by using the convolution kernels to scan an entire image, relatively few parameters need to be learned compared to the total number of operations. Their adoption has exploded in the last few years because of two recent developments. First, large, labeled data sets such as the Large Scale Visual Recognition Challenge (ILSVRC) [3] have become available for training and validation. Second, CNN learning algorithms have been implemented on the massively parallel graphics processing units (GPUs) which tremendously accelerate learning and inference.

To achieve accurate results, CNNs need many parameters (over 100M parameters reported in [4]) and require huge amounts of computational resources and memory, they also offer significant potential for massive parallelization and extensive data reuse.

As a result, expensive and power-hungry accelerators as GPUs are needed to efficiently process these networks, recently many applications such as embedded systems in self-driving cars need high energy efficiency and real-time performance. Therefore, there is a need to reduce the computational resources to reduce the used power and speed up the calculations.

FPGA implementations of CNN have seen an increased amount of interest in recent years due to the customizability of FPGAs; Designers can create dedicated pipelines with parallel processing elements, customized bit width, etc. on FGPAs. Therefore, there is a rapid increase in popularity of using FPGAs as accelerators. They also have advantages of good performance, high energy efficiency, fast development round, and capability of reconfiguration.

On the other hand, the limitations of the computational resources and memory bandwidth of an FPGA platform must be considered. In fact, if an accelerator structure is not carefully designed, its computing throughput cannot match the memory bandwidth provided by the FPGA platform. It means that the performance is degraded due to the bottleneck of the memory bandwidth.

Both CNN algorithms and FPGA aggravate the problem of achieving the best performance. As a result, it's a must to seek ways in order to reduce the number of the computations and the energy consumption (specially at convolutional layers that require a huge number of operations) and match the computing throughput to the memory bandwidth (this problem specially appears at fully connected layer as its operations is simple and does not require much time so the latency would be caused from memory bandwidth). Acceleration approaches can be categorized into two main parts; (1) General approaches (hardware independent) as reduction in precision, Shared weights and data reuse. (2) Customize the FPGA architecture to be suitable for the algorithm using pipelining, parallelism and increase the memory bandwidth.

1.2. Problem statement

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. Also, deep-learning systems thrive on data. The more data an algorithm sees, the better it'll be able to recognize and generalize about, the patterns it needs to understand. 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 like smartphones, smart glasses, and other wearable devices.

As a result, hardware accelerators such as Graphic Processing Units (GPU), Field Programmable Gate Arrays (FPGA), and Application Specific Integrated Circuits (ASIC), have been utilized to improve the throughput of the CNN.

Among these accelerators, GPUs are the most widely used to improve both training and classification process of CNN, thanks to their high throughput and memory bandwidth. However, GPUs consume a considerable amount of power which is another important evaluation metric in the modern digital systems.

ASIC design, on the other hand, has achieved high throughput with low power consumption by assigning dedicated resources and customizing memory hierarchy. But

the development time and cost are significantly high compared to other solutions. As alternative, 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 [5].

The capacity of hardware resources in the FPGA increases continuously, which provides more than a thousand floating computing units in one FPGA chip and provide low power consumption. Also, CNN offers significant potential for massive parallelization and extensive data reuse which make FPGAs suitable for customizing the designs of CNNs to achieve low power consumption and high throughput.

One limitation for the design of efficient accelerators on FPGAs is the limited amount of external memory bandwidth. The current bottleneck in available platforms for efficient utilization of parallelism is data transfer as CNN requires large memory bandwidth due to FC layers. Without on-chip buffers all accesses are to the external memory, requiring huge memory bandwidth and consuming a lot of energy. The number of external accesses can be reduced by on-chip memory that exploits data reuse due to the heavily pipelined circuits in FPGA implementations [6].

The most challenging problem for CNN is the Real-time classification, that accepts live data input from different devices and satisfy the real-time performance requirements while constraining energy usage, because of the need to run the multiple layers of a convolutional neural network in real time (as in embedded computer platforms for autonomous cars which are expected to be one of the key beneficiaries of deep learning and neural networks) [7].

1.3. Solution approach

FPGAs are customizable and programmable to deliver low latency and flexible precision, with higher performance per watt for deep learning inference. In order to reduce the number of the computations and the energy consumption, several techniques of optimization and approximation can be used on FPGAs to accelerate the algorithm.

First is Data Reuse; As each input layer influences all output layers in a CNN convolution layer it is possible to process multiple input layers simultaneously. This would

increase the external memory bandwidth required for loading layers. The data is cached in FPGA memory allowing each pixel to be reused multiple times [8].

Second are approximate computing techniques:

I- The Precision Reduction

Typically, CNN run on high precision machines using 32-bit floating point number representations or 16-bit fixed point. However, such high precision is not always necessary. The energy spent in high precision computations, does not lead to more accurate classification by the algorithm. To reduce the energy consumption of the CNN's computations, the main strategy is to quantize its weights and the inputs to its layers. Such quantization leads to a network that is only an approximation of the original network. The unique flexibility of the FPGA fabric allows the logic precision to be adjusted to the minimum that a particular network design requires [9].

The reduction in precision allows the FPGA accelerator to process increasingly more images per second. This can be achieved in various ways, leading to either a static (fixed after design-time) or a dynamic (adaptable after design-time). The work in [9] shows how the reduction in precision can be done for different network architecture with minimal loss in accuracy and without the need to retrain the network which leads to reduce the energy consumption.

There are two types of the precision reduction as discussed in [9]

1- Uniform quantization which uses the same quantization setting (the number of quantization bits) for all the network layers.

2- Pre-layer quantization where each layer is quantized separately which lead to better results.

II- Pruning which is the mean of eliminating all relatively small weights of neurons, which decreasing a significant number of operations, resulting in a high throughput.

Third is computational skipping:

Due to the appearance of the Rectified Linear Unit layers (RELU layers) in many modern convolutional neural networks. These put all negative inputs to zero and pass on positive values unchanged, Output = max (0, Input). Since many layers in CNN classification algorithms only output positive values when certain features are present, a large amount of RELU outputs will be zero and do not have to be used for further computations. The RELU layers thus allow for additional energy reductions by not computing unnecessary computations through computation skipping [9].

1.4. Organization

The following chapters discuss the CNN algorithm structure, how it is implemented by software and hardware and comparison between the two approaches. The remainder of this thesis is organized as follows:

Chapter 2 provides background information on Neural Networks especially Convolution Neural Networks (CNN), discusses the main layers of the CNN with their equations and functions and provides information about the training process performed on any neural network to improve its accuracy. Then it discusses background on FPGAs, including a brief overview of FPGA architectures and provides a literature survey including some techniques for implementing the CNN on FPGAs and acceleration methods.

Chapter 3 provides information on the chosen CNN for the project which is Alex-Net having a quick overview on its accuracy, the number of its layers and their arrangement.

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.

Chapter 4 provides a discussion on the project's chosen design with the details of how each layer of the 5 main layers of Alex-Net is implemented and accelerated using two main techniques, parallelism of resources and pipelining inside some layers

Chapter 5 provides results of the synthesis and implementation of super layer1 (Convolutional+ Pool+ Normalization) in Alex-Net using Vivado showing the number of

resources utilized on the chosen kit which is Virtex 7, the timing constraints and the power consumption of the design on the kit.

Chapter 6 discusses a modification on the design discussed in chapter 4 showing how this affects the number of resources and the simulation time emphasizing the target of the project which is accelerating Alex-Net on FPGA.

Chapter 7 provides the results of the simulation of the whole design on Vivado and discusses the simulation timing results on MATLAB and Virtual FPGA (Behavioral Simulation).

Chapter 8 provides a brief overview of the findings, draws conclusions, and recommends directions for future work.

Chapter 2.
Background and Related work

2.1. Convolutional neural networks

2.1.1. Neural Networks overview

NN is a computational model inspired by the way we believe our brain operates: the data that comes from our sensors, e.g., eyes, is processed by multiple simple computational units called neurons. The neurons are interconnected through a complex network of connections (axons), and after several transformations, the input is translated into a conclusion such as “there is a chair in the picture” [10].

Similarly, artificial NNs use vast amounts of simple computational elements that are organized in interconnected layers called also neurons. 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 the input data, an NN will calculate the probability that the data belong to a certain class (e.g., an object in a specific image). A Neural network needs in order to work to be trained at first. The network can be trained to recognize different classes by being provided a set of labeled training data (data sets). For example, given a set of faces and a set of non-faces, it can learn to decide whether an image contains a face. This is called supervised learning. Training of the NN involves more computations and takes more time than using a network and will be discussed in section 2.1.4.

2.1.2. Convolutional Neural Networks overview

CNNs are a type of NN commonly used in image processing. Convolutions allow NNs to use the way information is structured in the image to reduce the number of calculations and improve feature extraction.

A typical CNN structure consists of a feature extractor and a classifier. The feature extractor extracts an input image’s features and sends them to the classifier. According to these features, the classifier decides the category that the input image belongs to. A feature extractor consists of several similar stages. The input and output of a stage are

called feature maps. The output feature maps of a stage are the input of the next stage. The input image is the input to the first stage. Each stage consists of three layers: a convolutional layer, a nonlinearity (RELU) layer, and a sub-sample (pooling) layer. The output feature maps of the last stage are organized as a feature vector of the original input image and sent to the classifier. A classifier is a traditional MLP (multi-layer perceptron) composed of several full connection layers. It takes the feature vector as input and calculates the probability of each category that the input image may belong to. At last, the classifier chooses the category with highest probability as the output [11].

2.1.3. Convolutional Neural Network Layers

Convolutional layer

The first layer in a CNN is a Convolutional Layer. Figure 2.1 illustrates the computation of a convolutional layer. 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. 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. By stacking a number of convolutional layers, the network hierarchically learns high-level features of the image [12].

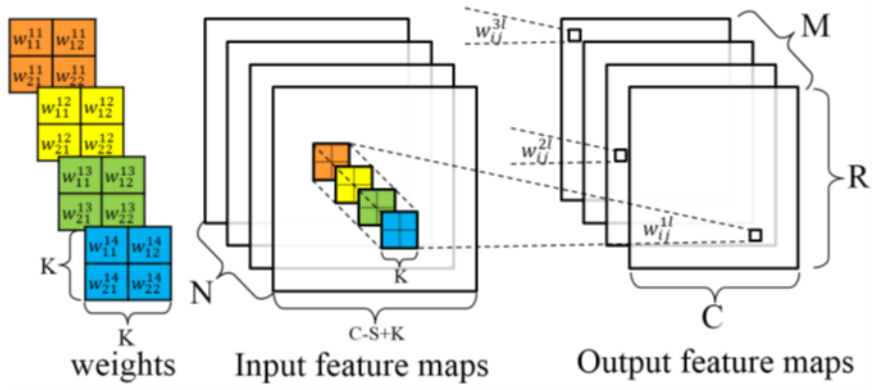


Figure 2.1 Convolution operation

RELU (Rectified Linear Units) Layer

After each convolutional 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 convolutional 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. The RELU layer applies the function $F(x) = \max\ (0, x)$ to all of the values in the input volume. In basic terms, this layer just changes all the negative activations to zero. This layer increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the convolutional layer.

Pooling layer

In a typical CNN, convolutional layers are interleaved with pooling layers. Pooling layers are used to reduce feature map dimensions by subsampling with some simple function; for example, 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. As shown in Figure 2.2, the filter size is $2\ x\ 2$ and the stride has the same length. A pooling layer serves two main purposes. The first is that the number of parameters or weights is reduced by 75% in the previous example, thus lessening the computation cost. The second is that it will control over-fitting. This term refers to when a model is so tuned to the training examples that it is not able to generalize well for the validation and test sets. A symptom of over-fitting is having a model that gets 100% or 99% on the training set, but only 50% on the test data.

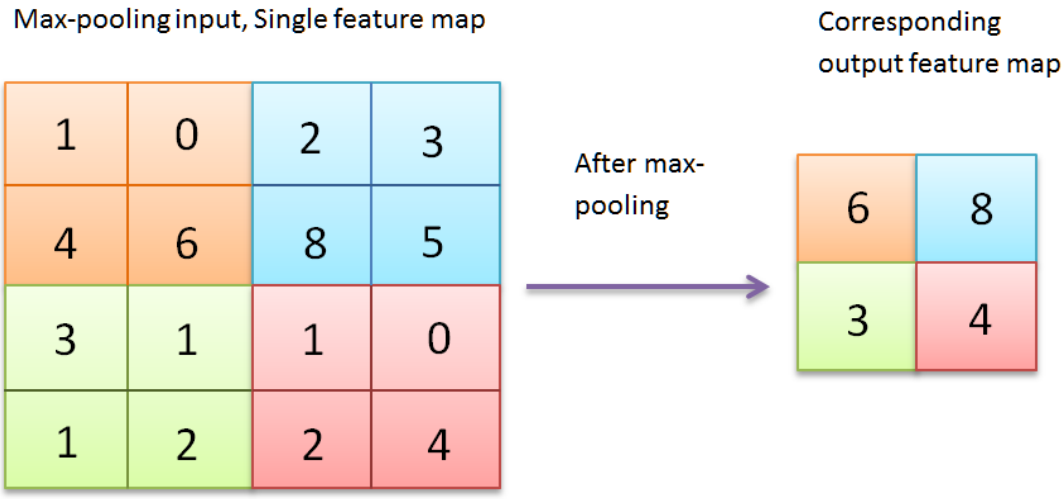


Figure 2.2 Max pooling

Local Response Normalization (LRN)

The Local Response normalization (LRN) reduces top-1 and top-5 error rates by 1.4% and 1.2%, respectively [13]. This sort of response normalization implements a form of lateral inhibition inspired by the type found in real neurons. Its contribution in the network performance was verified on the CIFAR-10 dataset: a four-layer CNN achieved a 13% test error rate without normalization and 11% with normalization [14].

$$out^i = in^i_{(x,y)} / (k + \propto \sum_{j=\max(0,i-\frac{n}{2})}^{\min(N-1,i+\frac{n}{2})} \left(in^j_{(x,y)}\right)^2)^\beta \qquad (1)$$

$$\mathrm{k} = 1, \mathrm{n} = 5, \alpha = 10^{-4}, \text{and } \beta = 0.75$$

Fully connected layers

The way this fully connected 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 then outputs an N dimensional vector where N is the number of classes that the program has to choose from. Figure 2.3 shows an example of FC layer with un-rolled input $1 \; x \; 3072$ and corresponding weights $3072 \; x \; 10$.

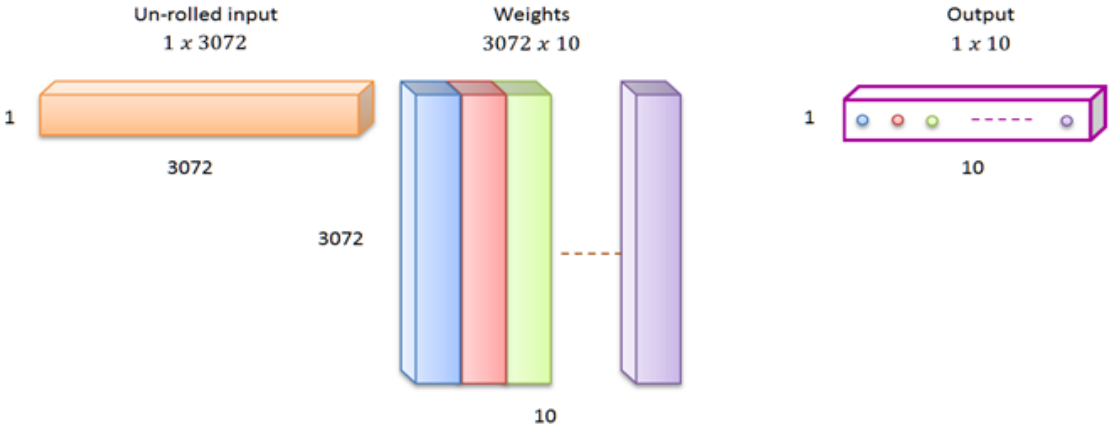


Figure 2.3 FC operation example

2.1.4. Training process

A CNN needs in order to work to go through a training process called back-propagation. Back-propagation can be separated into 4 distinct sections, the forward pass, the loss function, the backward pass and the weight update. During the forward pass, you take a training image (RGB image) which for example a 32 x 32 x 3 array of numbers and pass it through the whole network. For example, CNN categorize 10 classes corresponding to numbers from 0 to 9. On the first training example, since all of the weights or filter values were randomly initialized, the output will probably be something like [.1 .1 .1 .1 .1 .1 .1 .1 .1 .1], basically an output that doesn't give preference to any number in particular. The network, with its current weights, isn't able to look for those low-level features or thus isn't able to make any reasonable conclusion about what the classification might be. This goes to the loss function part of back-propagation. The training data has both an image and a label. Let's say for example that the first training image inputted was a 3. The label for the image would be [0 0 0 1 0 0 0 0 0 0]. A loss function can be defined in many different ways but a common one is MSE (Mean Squared Error), which is ½ times (actual - predicted) squared.

$E_{total} = \sum 0.5\ (target - output)^2$

The predicted label (output of the CNN) must be the same as the training label (This means that our 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 2.4.

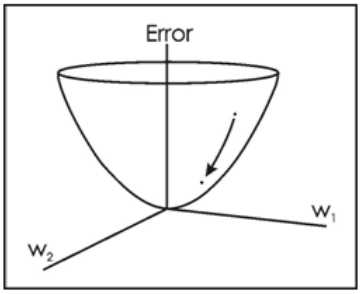


Figure 2.4 the error function

Now perform a backward pass through the network, which is determining which weights contributed most to the loss and finding ways to adjust them so that the loss decreases. Once this following derivative is computed, we then go to the last step which is the weight update, update all the weights of the filters so that they change in the opposite direction of the gradient.

$W = Wi - \boldsymbol{\eta}\frac{\boldsymbol{dL}}{\boldsymbol{dW}}$, *Where W is weights, L is loss function, and* $\boldsymbol{\eta}$ *is learning rate.*

2.2. FPGAs

2.2.1. Introduction

A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing. The FPGA configuration is generally specified using a hardware description language (HDL), Verilog or VHDL, similar to that used for an application-specific integrated circuit (ASIC).

2.2.2. Components

Field Programmable Gate Arrays (FPGAs) offer a reconfigurable design platform which makes them popular among digital designers. Typical internal structure of FPGA comprises of three major elements as shown in Figure 2.5[15]:

1. Configurable Logic Blocks (CLBs) are the resources of FPGA meant to implement logic functions. Each CLB is comprised of a set of slices which are further decomposable into a definite number of look-up tables (LUTs), flip-flops (FFs) and multiplexers (Muxes).

2. Input/output Blocks (IOBs) available at FPGA's periphery facilitate external connections. These programmable blocks carry signals 'to' or 'from' FPGA chip.

3. Switch Matrix is an interconnecting wire-like arrangement within FPGA. This offer connectivity for the CLBs or provide dedicated low impedance, minimum delay paths (for example, global clock line).

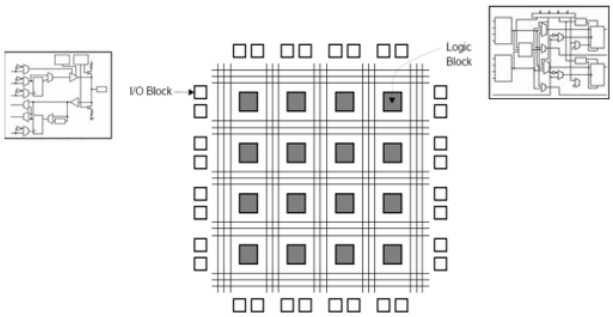


Figure 2.5 Internal structure of FPGA

Some other important resources in the FPGAs as shown in Figure 2.6:

1. Hardware multipliers denoted as digital signal processing (DSP) units which are used in MAC operations, multiplication and accumulation operations, which is widely used in Convolutional neural networks.

2. Block RAMs which are prepared in columns as shown in the figure. In Xilinx FPGAs the BRAMs can be in 2 sizes 18 Kb or 36 Kb. For example, if the module synthesized has size less than 18 Kb it`s synthesized as 18 Kb BRAM and if it`s between 18 Kb and 36 Kb, it`s synthesized as 36 Kb BRAMs.

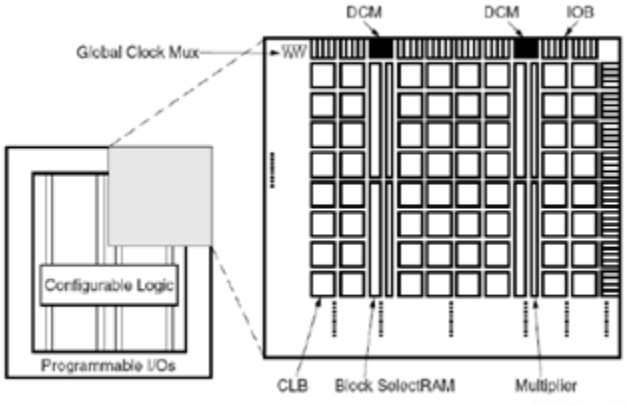


Figure 2.6 FPGA resources

2.2.3. Design Flow

Figure 2.7 shows the steps of the design flow [16].

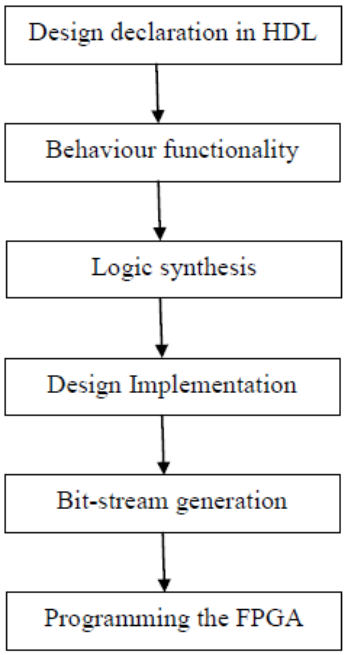


Figure 2.7 FPGA design flow

Explaining the previous steps

1. Design Entry: It can be subdivided into two phases: defining functionality and structure of the design and creating the design using Hardware Descriptive Language (HDL), Verilog or VHDL.

2. Behavioral functionality: Behaviorally simulating the HDL designs to test system and device functionality before synthesis.

3. Synthesis: Converts the input HDL source files into a netlist. It's divided into three-step process:

a) Syntax check & design association to logic cells.

b) Optimization: Reducing logic, eliminating the redundant one to make the design smaller and faster.

c) Technology Mapping: Connecting design to logic, predicting and adding timing estimates, creating output reports and generating netlist file containing the design and constraints.

4. Implementation: Determining the physical design layout by mapping synthesized netlists to the target FPGA's structure and interconnecting design resources to FPGA's internal and I/O logic. It consists of three sub-processes as shown in Figure 2.8:

a) Translate: Combining all netlists and constraints into one large netlist and pining assignment & time requirements (e.g. input clock period, maximum delay, etc.) provided via a User Constraints File.

b) Map: Comparing the resources specified in the input netlist file against the available resources of the target FPGA and dividing netlist circuit into sub-blocks to fit into the FPGA logic blocks generating a Native Circuit Description (NCD).

c)Place & Route (PAR): Placing physically the NCD sub-blocks into FPGA logic blocks and routing signals between logic blocks such that timing constraints are met generating a completely routed NCD file.

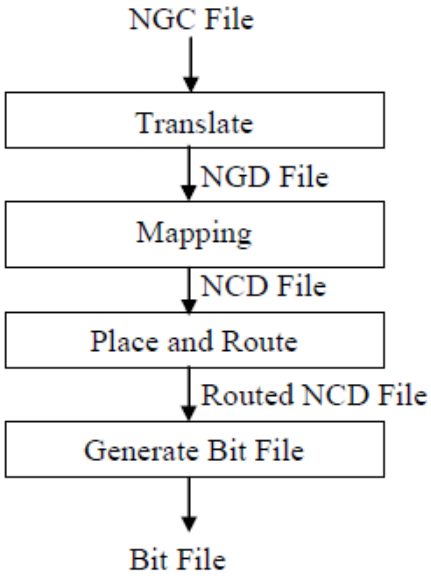


Figure 2.8 FPGA implementation steps

5. Bit Stream Generation: Converts the final NCD file into a format the FPGA understands.

6. Programming the FPGA with the generated bit stream.

2.3. Literature survey

Due to the specific, complex computation pattern of CNN, general purpose processors (CPUs) are not efficient for CNN implementation and can hardly meet the performance requirement. Thus, various accelerators based on FPGA, GPU, and even ASIC design have been proposed to improve performance of CNN designs [12]. Among these approaches, FPGA based accelerators have attracted more and more attention.

The authors of paper [17] consider whether future high-performance FPGAs will outperform GPUs for next-generation DNNs in terms of speed beside its superiority in power consumption-efficiency, evaluating a selection of emerging DNN algorithms on two generations of Intel FPGAs (ArriaTM 10, StratixTM 10) against the latest highest performance Titan X Pascal GPU. They study various GEMM operations for next-generation DNNs, and then proposed a detailed case study on accelerating Ternary Res-Net which relies on sparse GEMM on 2-bit weights (i.e., weights constrained to 0, +1, -1)

and full-precision neurons which its accuracy is within ~1% of the full precision Res-Net which won the 2015 Image-Net competition. The results were very promising; Stratix 10 performance is 10%, 50% and 5.4x better in performance (TOP/sec) than Titan X Pascal GPU on GEMM operations for pruned, Int6, and binarized DNNs, respectively. On Ternary-ResNet, the Stratix 10 FPGA is projected to deliver 60% better performance over Titan X Pascal GPU, while being 2.3x better in performance/watt. Results indicate that FPGAs may become the platform of choice for accelerating DNNs.

Not only FPGAs are recommended to be used as hardware accelerators in implementing CNN but also ASIC technologies which gives a better performance, on the other hand the FPGA based accelerators have attracted more attention of researchers than ASIC accelerators because they have advantages of quite good performance, fast development round and capability of reconfiguration.

Authors of [18] consider the spatial architectures used in ASIC and FPGA-based accelerators, discussing how data-flows can increase data reuse from low cost memories in the memory hierarchy to reduce energy consumption. This includes a large global buffer with a size of several hundred kilobytes that connects to DRAM, an inter-PE network that can pass data directly between the ALUs, and a register file (RF) within each processing element (PE) with a size of a few kilobytes or less. They investigate data-flows that exploit three forms of input data reuse (convolutional, feature map and filter). For convolutional reuse, the same input feature map activations and filter weights are used within a given channel, just in different combinations for different weighted sums. For feature map reuse, multiple filters are applied to the same feature map, so the input feature map activations are used multiple times across filters. Finally, for filter reuse, when multiple input feature maps are processed at once (referred to as a batch), the same filter weights are used multiple times across input features maps.

Authors of [19] proposed energy-efficient dataflow called row stationary, which aims to maximize the reuse and accumulation at the local memory level (register file or caches) for all types of data (weights, pixels and partial sums) for the overall energy efficiency. It keeps the row of filter weights stationary inside the RF of the PE and then streams the input activations into the PE. The PE does the MACs for each sliding window at a time, which uses just one memory space for the accumulation of partial sums. Since there are overlaps of input activations between different sliding

windows, the input activations can then be kept in the RF and get reused. By going through all the sliding windows in the row, it completes the 1-D convolution and maximizes the data reuse and local accumulation of data in this row. With each PE processing a 1-D convolution, multiple PEs can be aggregated to complete the 2-D convolution.

Also, authors of [18] idly discuss how DNN models and hardware can be co-designed to jointly maximize accuracy and throughput, while minimizing energy and cost, which increases the likelihood of adoption. They highlight various efforts that have been made towards the co-design of DNN models and hardware. The co-design approaches can be loosely grouped into the following categories: (1) Reduce precision of operations and operands; this includes going from floating point to fixed point, reducing the bit-width, non-linear quantization and weight sharing, (2) Reduce number of operations and model size; this includes techniques such as compression, pruning and compact network architectures.

The previous papers cited were the inspiration of our proposed approach, which applies feature-map data reuse technique, 2 data-flows types: output stationary and no local reuse, based on fixed point operations, uses extensively local memories for overall energy efficiency and then our CNN is accelerated on FPGA. Our approach will be deeply discussed and cleared in chapter4.

2.4. 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 the training process with all its stages performed on any neural network to improve its accuracy. Then it discusses background on FPGAs, including a brief overview of FPGA architectures and provides a literature survey including some techniques for implementing the CNN on FPGAs, acceleration methods such as parallelism, data reuse, …. etc. showing the results of this implementation as execution time of each layer on FPGA and power consumption.

Chapter 3.
Alex-Net

The chosen CNN architecture is Alex-Net, which had a large impact on the field of machine learning, specifically in the application of deep learning to machine vision. Alex-Net will be overviewed in this chapter and discussed from the software point of view.

3.1. Overview

Alex-Net is one of the state-of-the-art CNN, it won the 2012 ILSVRC (Image-Net Large-Scale Visual Recognition Challenge). It was the first model to achieve top-1 and top-5 error rates of 37.5% and 17.0% respectively on the test data of Image-Net dataset [14], which was an astounding improvement compared with the other top models in the context which had error rates of 28% and 26% [13] as shown in Figure 3.1, so this architecture was one of the first deep networks to push ImageNet Classification accuracy by a significant stride in comparison to traditional methodologies. Therefore, the network was the breakthrough of CNNs, and the reason to use CNNs in computer vision community.

In general, Neural networks are inspired by the structure of the cerebral cortex. At the basic level is the perceptron, the mathematical representation of a biological neuron. Like in the cerebral cortex, there can be several layers of interconnected perceptron. Input values, or in other words the underlying data, get passed through this “network” of hidden layers until they eventually converge to the output layer. The output layer is the prediction: it might be one node if the model just outputs a number, or a few nodes if it’s a multiclass classification problem.

The neural network developed by Krizhevsky, Sutskever, and Hinton in 2012, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of them are followed by max-pooling layers, and three fully-connected layers with a final 1000-way soft-max [25] as shown in Figure 3.2.

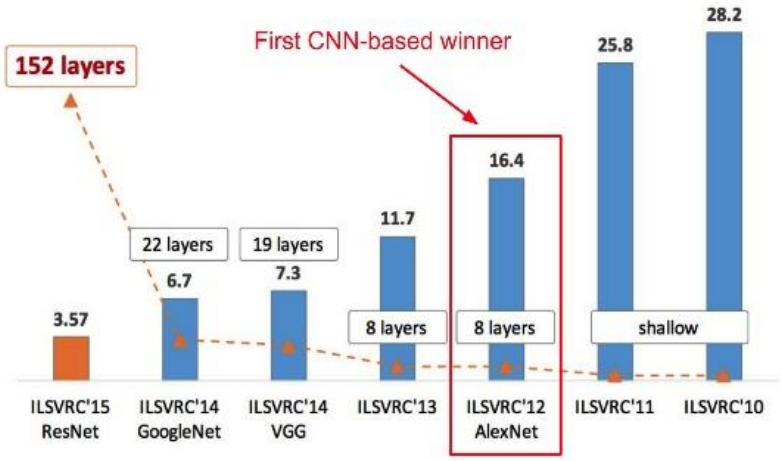


Figure 3.1 ImageNet Large Scale Visual Recognithion Challenge winners

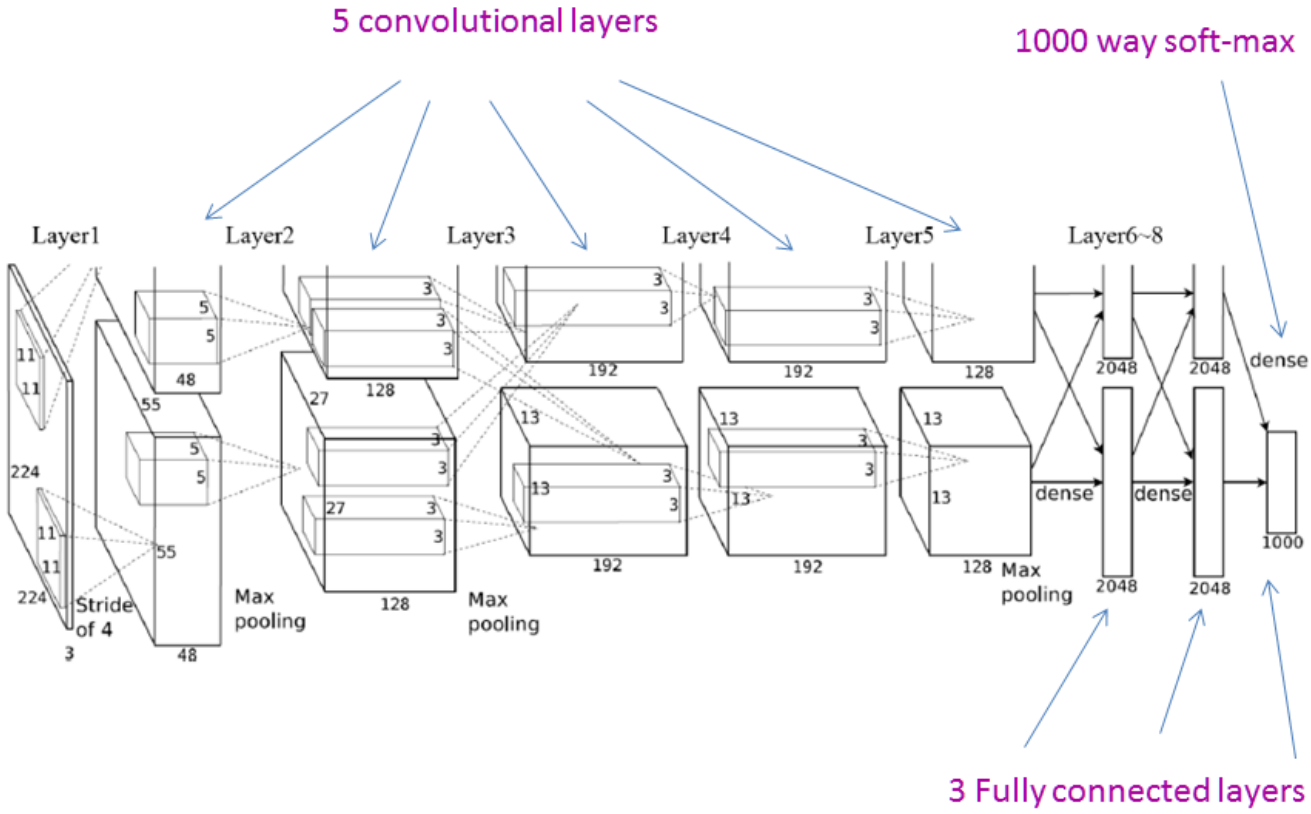


Figure 3.2 Alex-Netneural network architecture

3.2. The Network Architecture

Alex-Net contains 5 convolutional layers, max-pooling layers, dropout layers, and 3 fully-connected layers as shown in Figure 3.3. The used layout is a relatively simple layout, compared to modern architectures. The network was designed for classification with 1000 possible categories. Table 3-1 shows the detailed parameter in each layer of the network from Caffe.

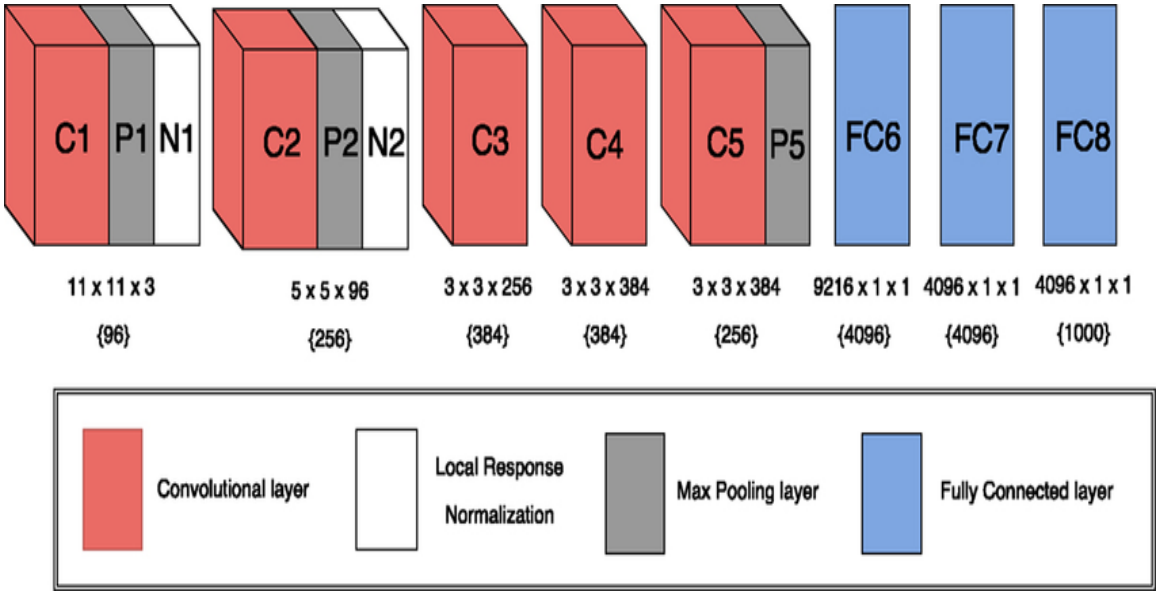


Figure 3.3 Alex-Net layers architecture

Table 3-1 The network parameters

Layer	1	2	3	4	5	6	7	8
Type	conv+ max+ norm	conv+ max+ norm	conv	conv	conv+ max	fc	Fc	fc
Channels	96	256	384	384	256	4096	4096	1000
Filter Size	11*11	5*5	3*3	3*3	3*3	-	-	-
Convolution Stride	4*4	1*1	1*1	1*1	1*1	-	-	-

Pooling Size	3*3	3*3	-	-	3*3	-	-	-
Pooling Stride	2*2	2*2	-	-	2*2	-	-	-
Padding Size	2*2	1*1	1*1	1*1	1*1	-	-	-

3.2.1. Convolutional Layer

The Conv layer is the core building block of a Convolutional Network that does most of the computational heavy lifting. It's always the first layer in a CNN.

- Convolution operation

The metrical convolution operator is applied over the feature maps of the input and filter as shown in Figure 3.4. 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.

The computation is given in (2), where M is the number of output feature maps (number of filters) of size E × E, C is the number of channels in Input feature maps, and R × R is the size of the Filter, which is the convolution operand obtained from the training.

$$out[m][h_o][w_o] = b_i + \sum_{i=1}^{C} \sum_{K_h=0}^{R} \sum_{K_w=0}^{R} IN\,[i][h_o + k_h][w_o + k_w] * Kernel[m][i][k_h][k_w] \quad (2)$$

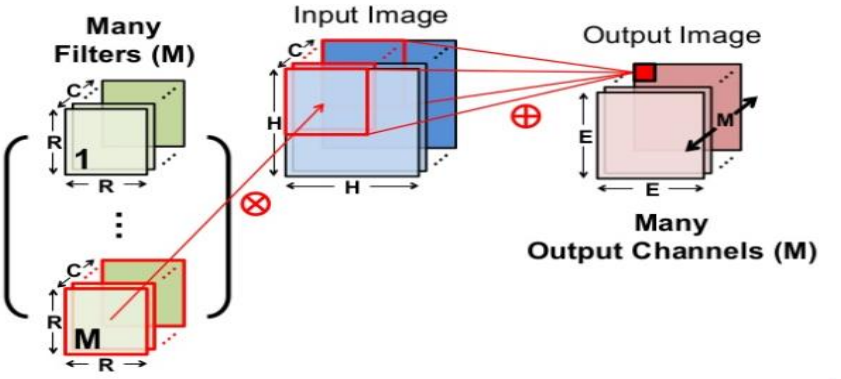


Figure 3.4 Convolution operation [19]

Convolution layer parameters:

- Filters [20]:

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.

- Stride [20]:

Stride is the number of pixels by which the filter matrix slides over the input matrix as shown in

Figure 3.5.

The spatial size of the output volume can be computed as a function of the input volume size (W), the filter size (K), the stride with which they are applied (S), and the amount of zero padding used (P) on the border.

The formula for calculating the output volume is given by (W−K+2P)/S+1.

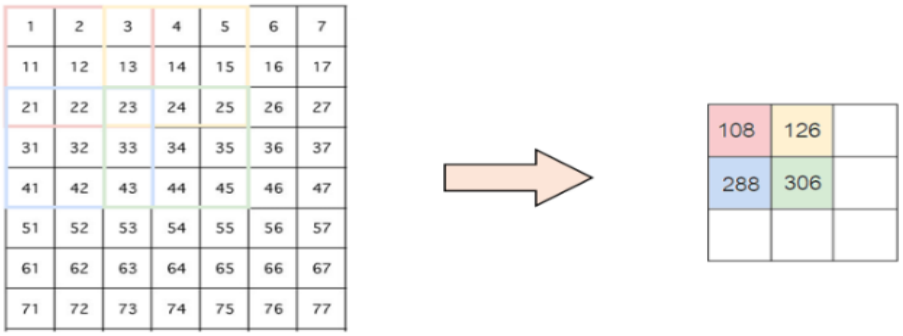


Figure 3.5 Filter Stride over 2-D input feature map

3.2.2. Pooling Layer

A key aspect of Convolutional Neural Networks are pooling layers, typically applied after the convolutional layers. Pooling layers (also called subsampling or down sampling) reduces the dimensionality of each feature map but retains the most important information. Pooling can be of different types: Max, Average, Sum etc. In practice, Max Pooling has been shown to work better [21]. For example,Figure 3.6 shows max pooling for a 2×2 window. Pooling operation is applied separately to each feature map.

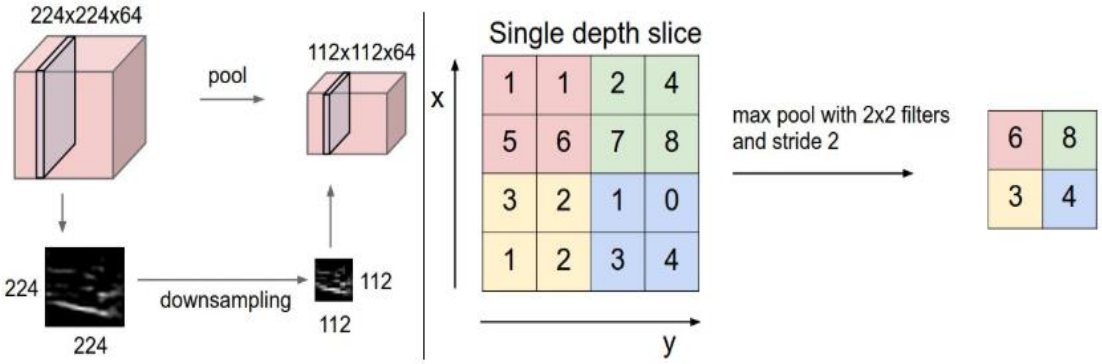


Figure 3.6 Max pooling operation

The function of Pooling is to progressively reduce the spatial size of the input representation [20]. In particular, pooling

- Makes the input representations (feature dimension) smaller and more manageable.
- Reduces the number of parameters and computations in the network, therefore, controlling overfitting.
- Makes the network invariant to small transformations, distortions and translations in the input image (a small distortion in input will not change the output of Pooling – since it considers only the maximum / average value in a local neighborhood).
- Provides a fixed size output matrix, which typically is required for classification. For example, if the network has 1,000 filters and then apply max pooling to each, it will get a 1000-dimensional output, regardless of the size of the filters, or the size of the input.

3.2.3. RELU

RELU stands for Rectified Linear Unit and is a non-linear operation used after every convolution operation. Its output is given by: max (0, in) as in Figure 3.7. RELU is an element wise operation and replaces all negative pixel values in the feature map by zero. The purpose of RELU is to introduce non-linearity in the CNN after linear operation of convolution (element wise matrix multiplication and addition), since most of the real-world data the network required to learn would be non-linear. Nonlinearity makes it easy for the model to generalize or adapt with variety of data to best fit its representation as in Figure 3.8 and to differentiate between the outputs [22].

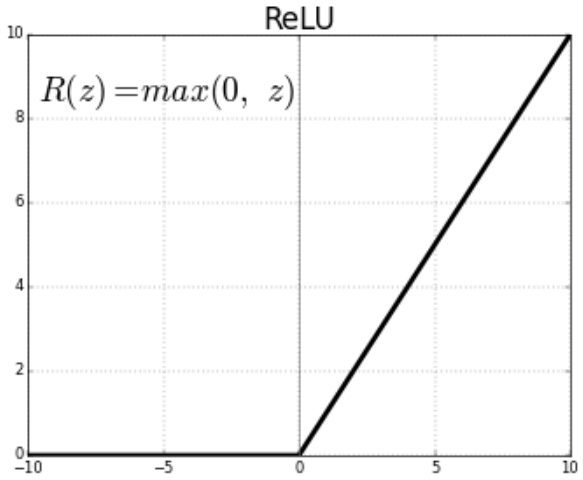


Figure 3.7 RELU function

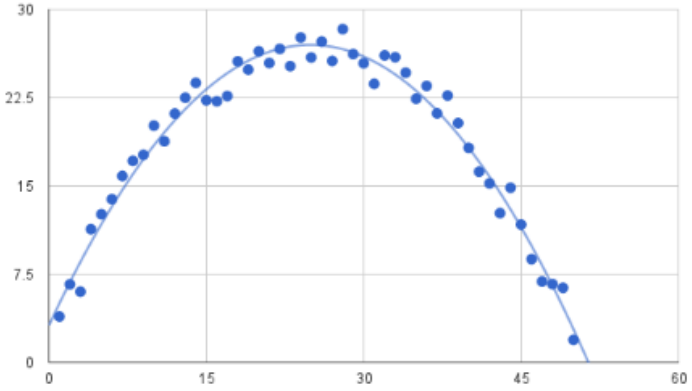


Figure 3.8 Non linear data fitting

There are other non-linear activation functions like Sigmoid and Tanh, however the RELU is used in almost all the convolutional neural networks or deep learning. it has become very popular in the last few years due to its advantages:

- It was found to greatly accelerate (e.g. a factor of 6 in [14]) the convergence of stochastic gradient descent compared to the sigmoid / tanh functions. It is argued that this is due to its linear, non-saturating form.

- Compared to tanh /sigmoid neurons that involve expensive operations (exponentials, etc.), the RELU can be implemented by simply thresholding a matrix of activations at zero [23].

3.2.4. Zero padding

Sometimes it will be convenient to pad the input volume with zeros around the border so that the filter can be applied to bordering elements of the input matrix as shown in Figure 3.9, as it doesn't have any neighboring elements to the top and the left. The size of this zero-padding is a hyper parameter. The nice feature of zero padding is that it will allow to control the spatial size of the output volumes [20].

0	0	0	0	0	0	...
0	156	155	156	158	158	...
0	153	154	157	159	159	...
0	149	151	155	158	159	...
0	146	146	149	153	158	...
0	145	143	143	148	158	...
...	...	...	...	...	...	...

Figure 3.9 Input matrix after zero padding

3.2.5. Local Response Normalization (LRN)

The role of the LRN layer is to normalize the RELU neurons which have unbounded activations to avoid the saturation in network, and subsequently data missing. Equation (3) shows the normalization functionality; normalizing around the local neighborhood of the excited neuron, and make it even more sensitive as compared to its neighbors.

$$b_{x,y}^{i} = a_{x,y}^{i} \Big/ \left(k + \alpha \sum_{j=\max\left(0,\ i-\frac{n}{2}\right)}^{\min\left(N-1,\ i+\frac{n}{2}\right)} \left(a_{x,y}^{j}\right)^{2} \right)^{\beta} \qquad (3)$$

The sum runs over n adjacent kernel maps at the same spatial position where,

$a_{x,y}^{i}$ Represents the *i* th Pool kernel's output at the position of (x, y) in the feature map.

$b_{x,y}^{i}$ Represents the output of local response normalization, and it's also the input for the next layer.

N is the number of the Pool's kernels (depth size).

n is the adjacent conv. kernel number; this number is up to you. In this network, n = 5.

k, α and β are hyper-parameters, whose values are determined using a validation set in the used pre-trained CNN; the values used k=1, n=5, α=0.0000, β=0.75.

Figure 3.10 illustrates the process of LRN in CNN, considering the following hints,

- This figure presumes that the *i* th kernel is not at the edge of the kernel space. If i equals zero or one or last or one to the last, one or two additional zero padding Conv kernels are required.
- In our network, n is 5, we presume n/2 is integer division, 5/2 = 2.
- Summation of the squares of output of RELU and Pool stands for: for each output of RELU and Pool, compute its square, then, add the 5 squared value together. This process is the summation term of the formula.

- I presume the necessary padding is used by the input feature map so that the output feature maps have the same size of the input feature map, if you really care. But this padding may not be quite necessary.

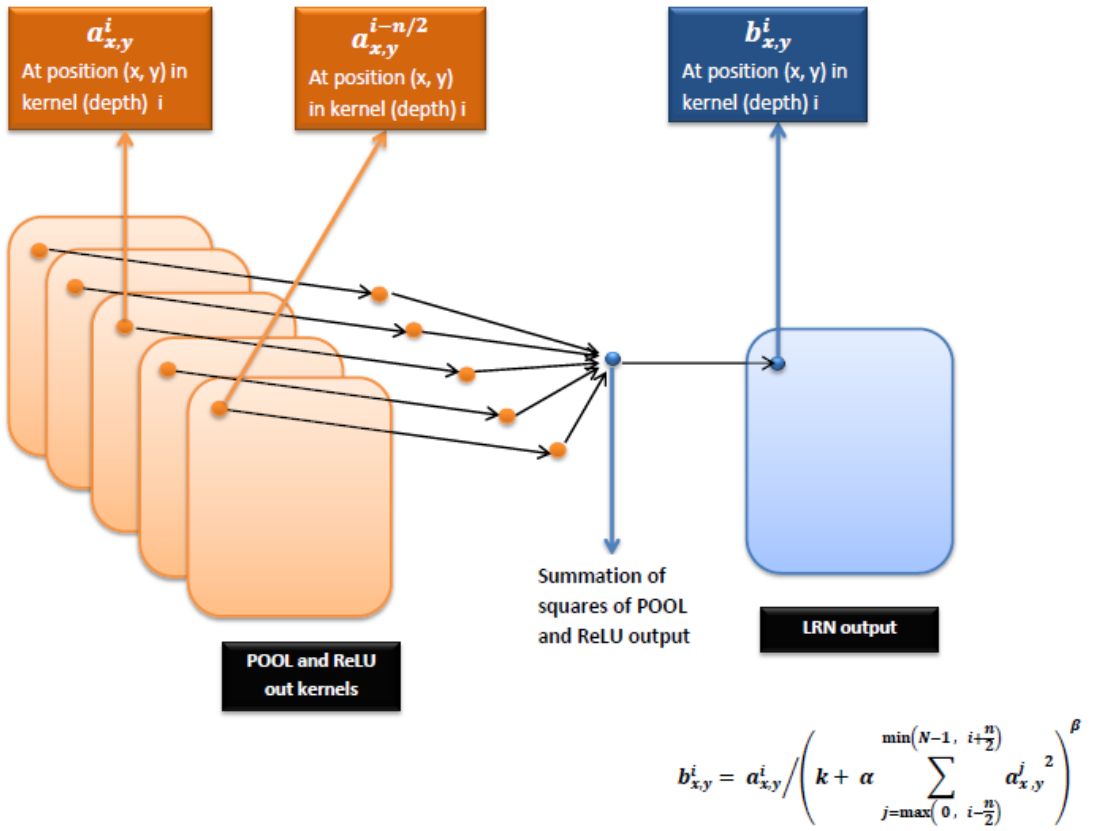


Figure 3.10 Normalization layer operation

The hardware implementation challenges can be observed from (3), it is arises in the fractional power (β=0.75), and the variable range of the summation loop. The fractional power (β=0.75) was rounded to (β=1) in the hardware implementation with slight reduction in the accuracy, the variable range of the loop will be implemented using FSM (finite state machine) as discussed in the following chapter.

3.2.6. Fully connected layer

The fully connected layer is a traditional Multi Layer Perceptron (MLP) with Soft-max activation function, its purpose is to use the extracted feature from the previous

convolution and pooling layers and classify the input image to various classes by determining the correlation between the extracted feature and a particular class.

The Multi Layer Perceptron contains one or more hidden layers with all the neurons connected to each other, which can learn linear or non–linear functions, Figure 3.11 shows a three FC layers with one input layer, one output layer and one hidden layer [24].

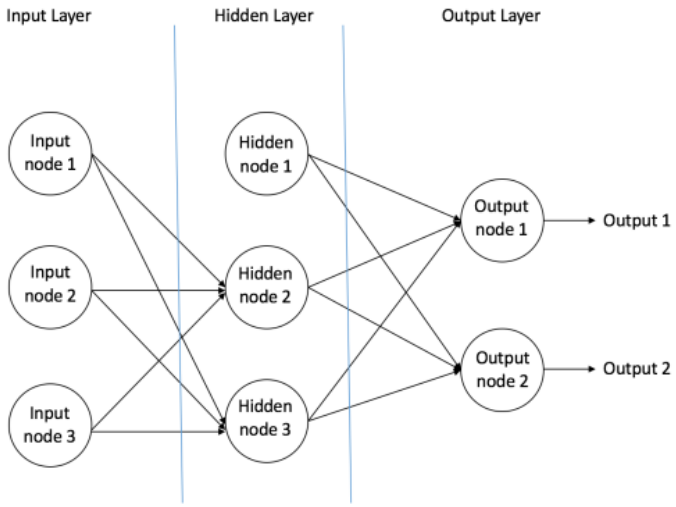


Figure 3.11 A multi layer perceptron with one hidden layer

The input FC layer unroll the output of the previous layer and multiply each neuron in the output by its corresponding weight as shown in Figure 3.12 using matrix multiplications in equation (4), the neurons of the output layer is fed to the Soft-max layer.

$$f(x, w) = x^T * w \quad (4)$$

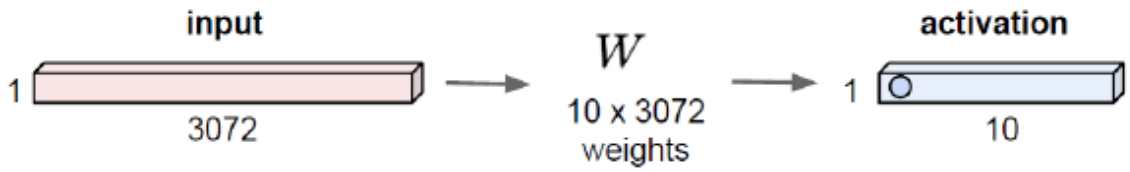


Figure 3.12 the fully connected layer using matrix multiplication

3.2.7. Soft-max

The last layer in the architecture is Soft-max. The Soft-max function squashes the outputs of each unit from real arbitrary values into some real values in range between 0 and 1 and sum up to one as illustrated in Figure 3.13, which represent the probability distribution of each category in the dataset. It can be interpreted as the (normalized) probability assigned to the correct label y_j given the image x_j and parameterized by W and the result is f_j given in equation (5) [23].

$$P(y = j|\, x) = e^{f_{y_j}} \Big/ \sum_{k=1}^{k=K} e^{f_k} \ , \ j = 1,2, \dots \dots \dots K, (5)$$

$$where\ f_j = x^T w\ and\ K\ is\ the\ total\ number\ of\ classes\ in\ the\ dataset.$$

Figure 3.13 The Softmax normalized probability.

Practical issues: Numeric stability. When writing a code for computing the Soft-max function in practice, the intermediate terms $e^{f_{y_j}}$ and $\sum_{k=1}^{k=K} e^{f_k}$ may be very large due to the exponentials. Dividing large numbers can be numerically unstable, so it is important to use a normalization trick. Notice that by multiplying the top and bottom of the fraction by a constant C and push it into the sum, it gives the following (mathematically equivalent) expression in (6). The constant C can take any value without affecting the results, a common choice for C is to set $\log C = -\max(f_j)$ to improve the numerical stability of the computation. This simply states that we should shift the values inside the vector f so that the highest value is zero the final equation is given by (7) [13].

$$\frac{e^{f_{y_j}}}{\sum_k e^{f_k}} = \frac{C\, e^{f_{y_j}}}{C \sum_k e^{f_k}} = \frac{e^{f_{y_j} + logC}}{\sum_k e^{f_k + logC}} \quad (6)$$

$$P(y = j|\, x) = e^{f_{y_j} - Max(f_j)} \Big/ \sum_{k=1}^{k=K} e^{f_k - Max(f_j)} \;,\; j = 1,2, \dots \dots \dots K, (7)$$

$$where\ f_j = x^T w\ and\ K\ is\ the\ total\ number\ of\ classes\ in\ the\ dataset.$$

The implementation challenge for this layer rises from the exponential and division, the exponent will be implemented using LUT and the division will be ignored as its function is to normalize the probability which will not affect the classifier decision.

3.2.8. Drop out

Deep neural network with multiple layers and large number of neurons suffer from over fitting during the training phase. The architecture consists of 5 convolution layers and 3 fully connected layers with 650,000 neurons make the usage of generalization techniques is necessary to prevent the network from over fitting. The dropout is a power full generalization technique for reducing overfitting in neural networks by preventing complex co-adaptations on training data by randomly dropping out the output of each hidden neuron with probability 0.5 as shown in Figure 3.14, which gives major improvements over other regularization methods.

The main idea of dropout is to have neuron A and neuron B both to learn something about the data, and the neural network not rely on 1 neuron alone as illustrated in Figure 3.15. This has the effect of developing redundant representations of data for prediction by randomly dropping out the outputs of the previous layer with probability of 0.5. However, there is have no idea which one is better, so in the testing phase, all the neurons are used but their outputs are multiplied by 0.5 to average them [25][26]. The dropout layer is used in the first two fully-connected layers of the network. Without dropout, the network exhibits substantial overfitting [25].

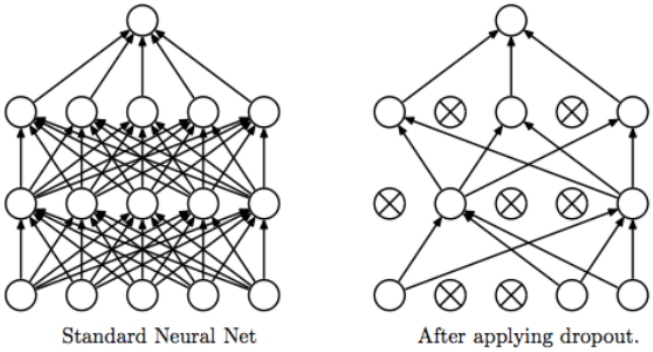


Figure 3.14 Droupout for multiple neurons in hidden layer

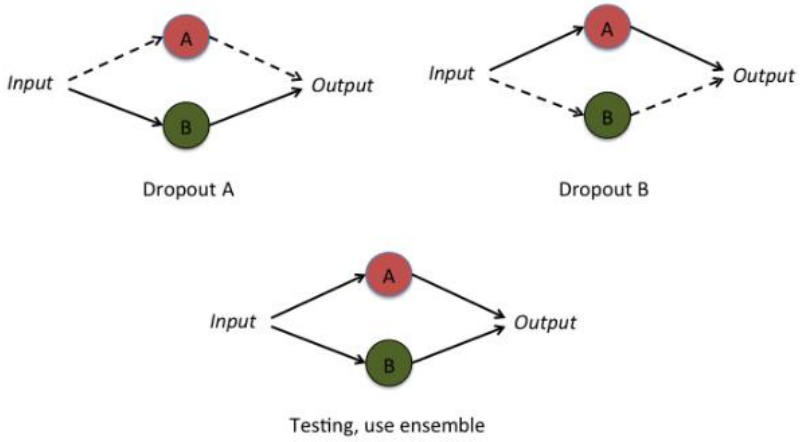


Figure 3.15 Droupout illustration for single layer with two neurons

3.3. Fixed point back ground

Fixed-point optimization of deep neural networks plays an important role in hardware based design and low-power implementations. In recent years increasingly complex architectures for deep convolution networks (DCNs) have been proposed to boost the performance on image recognition tasks. However, the gains in performance have come at a cost of substantial increase in computation and model storage resources. Fixed point implementation of DCNs has the potential to alleviate some of these complexities and facilitate potential deployment on embedded hardware.

A fixed-point representation of a number consists of integer and fractional components and sign bit as shown in Figure 3.16, 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}.

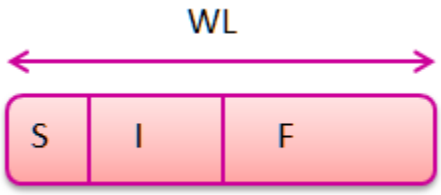


Figure 3.16 fixed point data representation

3.3.1. 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 the multiplicand has WL = 8, I=3 and F=4, and the multiplier has WL = 8, I=5 and F=2,

Multiplicand = $6.5625|_{\text{decimal}} = 0\ 110\ 1001|_{\text{fixed point representation}}$

Multiplier = $4.25|_{\text{decimal}} = 0\ 00100\ 01|_{\text{fixed point representation}}$

01101001

x 00010001

———————

01101001

00000000

00000000

00000000

01101001

00000000

00000000

00000000

000011011111001 = 0000011011.111001 = 27.890625

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

3.3.2. Fixed point addition

Addition is a little more complicated because the points need to be aligned before performing the addition. Using the same numbers from the multiplication problem,

0110.1001

+ 000100.01

———————

001010.1101 = 10.8125

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 $WL = WL + log_2\,(N\,x\,WL) + log_2\,(N)$. where N is the number of elements being summed.

3.4. Software Accuracy and preprocessing

A typical CNN is composed of two components: a feature extractor and a classifier. The feature extractor is used to extract the features of the input feature map by filtering it

with different filters. The feature extractor may consist of several convolutional layers followed by optional subsampling layers, for example the chosen network has five convolutional layers each layer has different number of filters with different sizes each filter concerned to extract certain features of the input (edges, corners, lines, etc. ……). The extracted features are then fed to the classifier, which is usually an artificial neural network, which is responsible to decide the category that the input image may belong to, according to the matching between the input image and each category in the dataset. The chosen network has three fully connected layers followed by soft-max, the function of each layer is discussed in section 3.2.

The chosen dataset for testing the network performance is subset of ImageNet, which was used in ILSVRC, it is a large dataset consist of 1000 categories and roughly 1.2 million training images, 50,000 validation images, and 150,000 testing images [25].

The network employs two paths, a feedforward path for recognizing the input image and a backward path for training. Before starting classifying the dataset the network must be trained on the training set to tune the network parameters, the training set of ImageNet has 1.2 million image, so training the network may take few months. Typically, the training process is done offline and the forward path only implemented on the FPGA. In order to save the training time on the software, a pretrained model on ImageNet from Caffe [27] is used, where the network parameters were available.

The network performance was tested by calculating the accuracy on the validation dataset of the ImageNet that contains 50,000 validation images. The test set of ImageNet wasn't used as the test set labels are available only for the competition submissions, although ILSVRC-2010 test set labels are available, it contains 150,000 testing images, which may take a month to classify it.

3.4.1. Accuracy

The accuracy of the network was tested on the cross-validation dataset from [28] using the open source MATLAB implementation of the forward path from GitHub [29]. The MATLAB implementation convert the input picture from MATLAB representation (RGB) to Caffe representation (BGR) and provide the preprocessing required for the input image and the implementation of each layer in the network.

The accuracy was measured on the first 1500 images from the data set for different number of bits represent data across the network.

- For the ideal representation of the data using 64 double data type the accuracy was about 57%.
- After replacing the power of normalization layer (β) (from 0.75 to 1) accuracy was about 53.67 %
- For 32-bit fixed point with 12-bits for integer and 20-bits for fraction part the accuracy reached 53.6%.
- For 16-bit fixed point with 12-bits for integer and 4-bits for fraction part the accuracy reached 53.5%.

3.4.2. Preparing data for Hardware implementation

In order to prepare the input data for the hardware simulation the data was required to fit into Block Ram which implies perform the operation on one input at a time, therefore the input data is unrolled from 3-D (RGB) representation to the 1-D representation to fit into the input ram. Typically, the input image after preprocessing is adjusted to 227x227x3, after the unrolling the generated output is of size 154587x1.The unrolling of data was done across the depth Figure 3.17 shows the unrolling of 3x3x3 input image as an example for simplicity. Figure 3.18 shows the flow to generate the input data to from MATLAB to the hardware simulation.

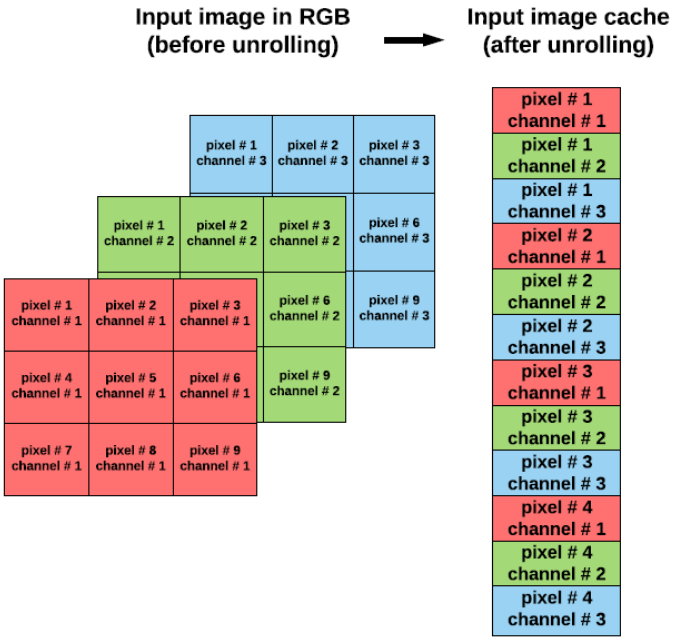


Figure 3.17 Unrolling the input data from 3-D to 1-D

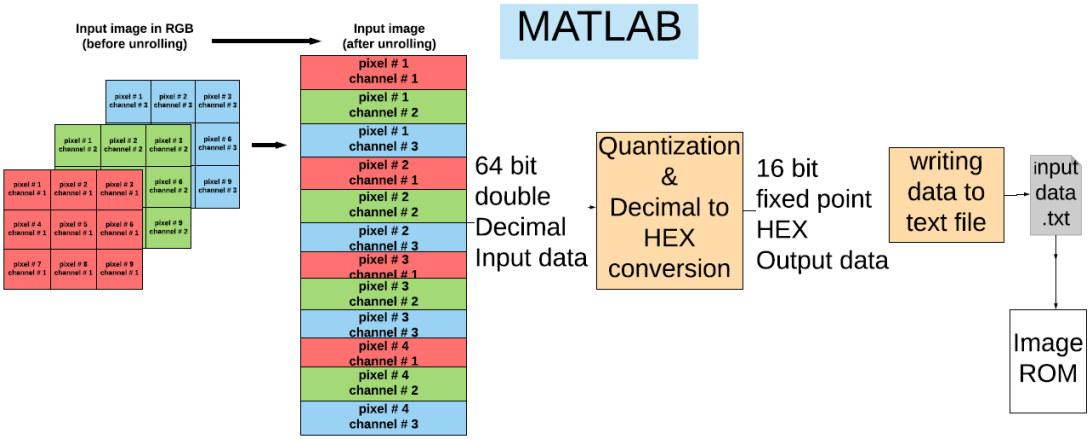


Figure 3.18 Data generation from MATLAB to Hardware simulation

3.5. Summary

This chapter provides information on the chosen CNN for the project which is Alex-Net having a quick overview on its accuracy for the top-5 and top-1 results using ImageNet, the number of its layers which is 13 layers (5 Convolutional layers, 3 Max Pooling layers, 3 Fully Connected layers and 2 Local Normalization layers) and their arrangement. 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. Proposed approach

4.1.1. Design

Our proposed approach is mainly based on serving convolutional layers as a previous study [18] proved that convolution operations will occupy over 90% of the computation time. It applies:

1- Usage of local memory hierarchy.

2- Data reuse technique; Feature map reuse.

3- Data flow techniques; (1) Output stationary and (2) No local reuse.

4- Fixed point operations and quantization.

Each point will be illustrated clearly in the rest of this section.

Usage of local memory hierarchy.

This design considers only one picture to be classified not involving a real-time sequence of image, that must be clear before involving the design techniques. All data types (weights, bias, one picture and output data of each layer) are stored in individual small local memories. Each filter kernel of weights is stored separately in caches and proposed to remain fixed in the whole design, and the picture is also stored in separate cache. Then output cache of each layer is needed and it represents the input cache of each layer.

So the CNN won't handle the external (off-chip DRR) memory except in loading state at the beginning and will store the data in these mentioned caches. That may introduce large area, but also will decrease dramatically the energy consumption, as low-level memories (caches) which have much smaller area than the DRR memory consumes much lower energy consumption in read and write operations.

Data reuse technique: Feature map reuse

For feature map reuse, multiple filters are applied to the same feature map, so the input feature map activations are used multiple times across filters. So the data comes from feature maps won't be recalled again, as all filters are working parallel on this feature map.

Data flow techniques

Output stationary (OS)

The partial sums are accumulated inside each PE (processing element) till the output is ready and then store the final result in the output cache, which helps to further reduce the energy consumption of accessing partial sums.

No local reuse (NLR)

Memory hierarchy lowest level is the caches level, no register files inside the PEs and all the on-chip area is allocated to the caches to maximize the storage capacity.

Fixed point operations

All operations through the CNN are fixed point operations, so inputs are 16-bit fixed point numbers and after applying the operations the output quantized to 16-bit also and then stored in output caches.

Figure 4.1 illustrates the mentioned design points on the first convolutional layer. One common cache for input feature map outputs one data pixel each time corresponding to control unit address, applied to the parallel PEs inputs. Each filter kernel has individual cache outputs one weight element corresponding to the control unit address, applied to the second input of the parallel PEs. These data were stored as 16-bit fixed point numbers in caches. Each PE works parallel to another PEs and applies convolution fixed point operation and accumulates in internal register "363 times for Conv 1", till the output is ready. The output is quantized to 16 bits. Then the final output is store in parallel output caches corresponding to each PE. The convolution PEs will be discussed in details in section 4.2.

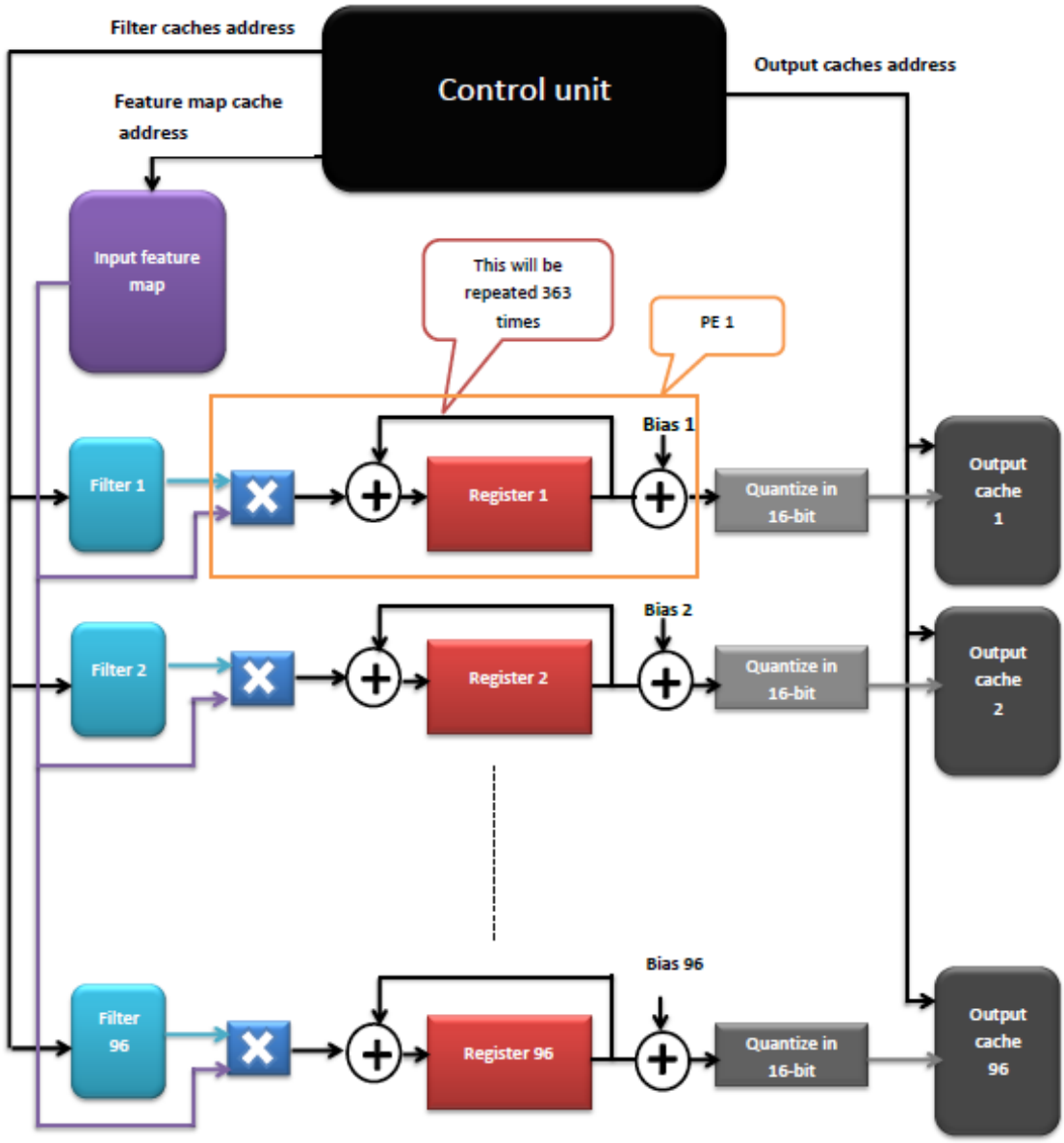


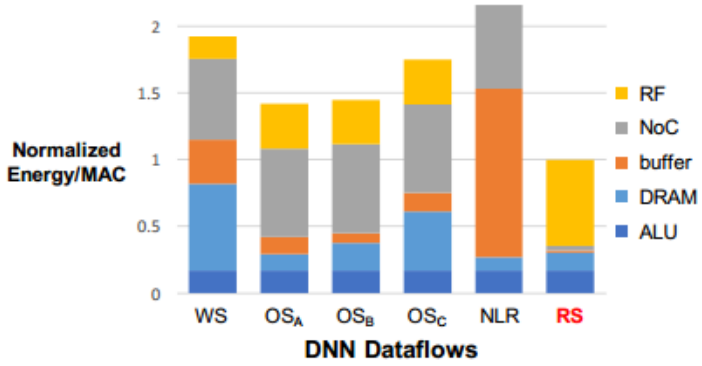
Figure 4.1 Hardware design over view

4.1.2. Comparative study

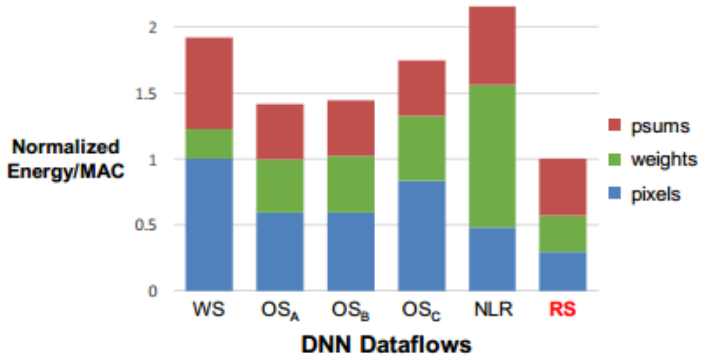
Authors of [18] discusses a comparative study between 3 different data flows; (1) Weight stationary, (2) Output stationary (A, B and C) and (3) Row stationary. The results were as shown in Figure 4.2. (a) showing that row stationary data flow gave the best performance in terms of energy efficiency. Our design has a lowest DRAM energy as it gets the data only one time from DRAM. It has relatively small buffer only for the bias shown in Figure 4.2 (a) so also a lowest buffer energy consumption. ALU equivalent to

the operation energy consumption as is the same as these data flows. No RF in our design only one register inside each PE. But our design introduces caches energy consumption which is relatively small as filter caches size is small and feature maps are reused and output cache won't be accessed unless the output is ready (no partial sums access times).

By comparing our design to results in Figure 4.2 (b), pixels will have the lowest energy consumption because of feature map reuse. Partial sums also the lowest energy consumption as we accumulate in internal register. weights are accessed more efficiently in row stationary data flow, but also in our design the local memory level will result in relatively lower or quite the same level of the row stationary weight-energy consumption.



(a) Energy breakdown across memory hierarchy



(b) Energy breakdown across data type

Figure 4.2 comparative study of energy consumption for different data flows

4.2. Convolution layer

Convolution layers form the crux of the CNN network, so it will be considered first to be explained. The mechanism of convolution layers as explained in section 3.3., is to

perform convolution of the input feature map with the layers' kernels with certain parameters as the stride, zero padding size, input feature map size, output feature map size and number of kernels which are different from one convolution layer to another (5 convolution layers), as shown in Figure 4.3.

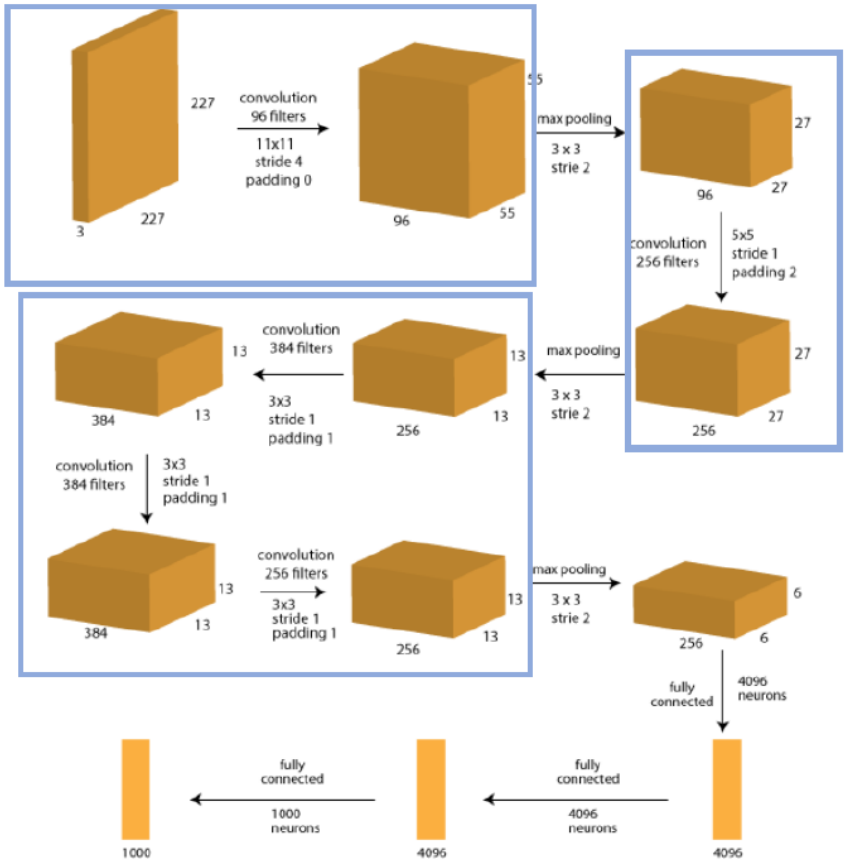


Figure 4.3 convolution layers in Alex-Net

Table 4-1 Alex-Net Convolutional Layers parameters

Layers	Input size	Output size	Number of filters	Filter size	Stride	Zero pad	Group
Conv1	227x227x3	55x55x96	96	11x11x3x96	4	0	1
Conv2	27x27x96	27x27x256	256	5x5x48x256	1	0	2
Conv3	13x13x256	13x13x384	384	3x3x256x384	1	1	1
Conv4	13x13x384	13x13x384	384	3x3x192x384	1	1	2
Conv5	13x13x384	13x13x256	256	3x3x192x256	1	1	2

As shown in Table 4-1, the convolution layers are the same expect for the parameters and the groups so the Conv1 layer will be explained in details.

Hardware implementation of Conv1 layer:

- The building unit is the parallel engine (PE) which is a multiplier and an accumulator as shown in Figure 4.4

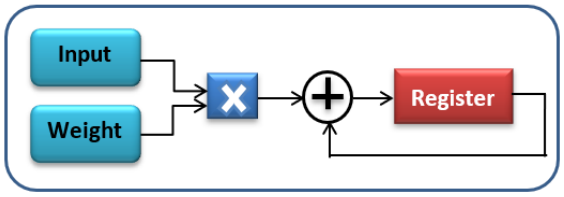


Figure 4.4 PE internal structure

- Then parallel PEs are used, number of PEs= number of filters.
  For Conv1 layers, the number of parallel PEs=96.
  So, the filters outputs are accumulated in parallel, while each filter convolution is done serially.
  Numbers of cycles taken for the Conv1 layer output to finish=number of outputs for each filter* number of cycles taken for each filter= 55*55*11*11*3.

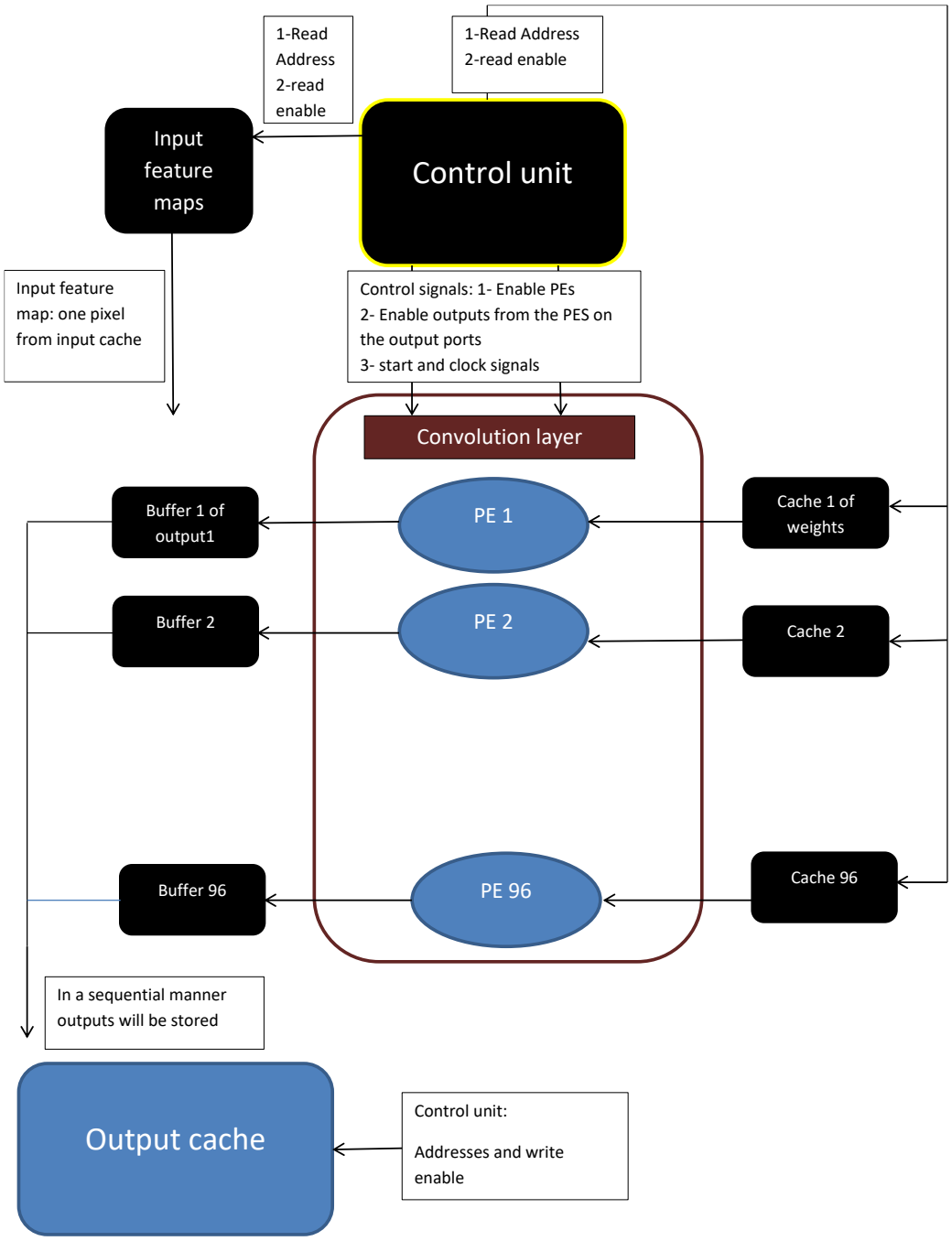


Figure 4.5 : Hardware structure of the parallel PEs, control unit, output caches, weights caches.

After the convolution output is ready, a bias is added to each filter output then the result passes through RelU layer (as discussed in section 3.2.3) and quantized into 16-bits and stored to be proceeded in the next pooling layer.

4.3. Pooling

After discussing the convolutional layer, the second layer is pooling. Figure 4.6 shows the basic building block of the pooling layer which mainly depends on comparing each input in the kernel with the previous inputs and keeping the maximum number in the register.

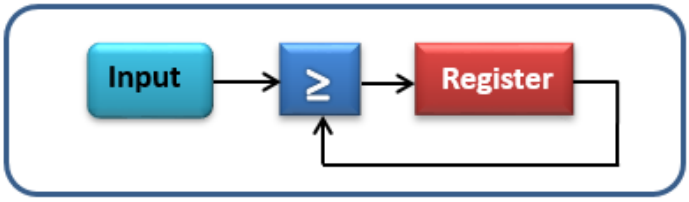


Figure 4.6 pooling engine

The main target of the project is acceleration of a network so parallelism is used to speed up the operations required from the network. The main idea of parallelism in this layer is using parallel engines of the building block corresponding to the depth of the input which means that each parallel engine is responsible for the output of a certain depth. Each PE has a separated input and output cache.

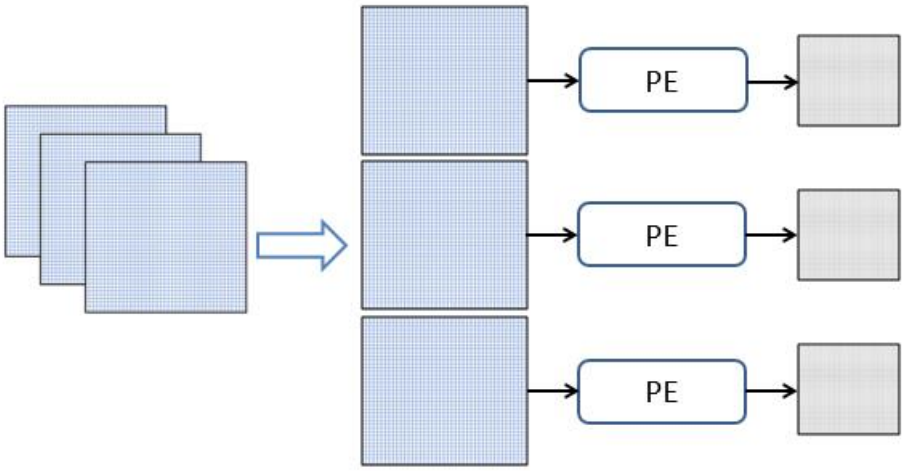


Figure 4.7 parallel engines for pooling layer

In Alex-Net, there exists three pooling layers so in the following table here are the

layers	Pool1	Pool2	Pool5
Number of Parallel Engines	96	256	256
Input Size of each Parallel Engine	55*55	27*27	13*13
Cache size Before Parallel Engine	4K * 4 bytes	1K * 4 bytes	256 * 4 bytes
Output Size of each Parallel Engine	27*27	13*13	6*6
Cache Size after Parallel Engine	1K * 8 bytes	256 * 8 bytes	64 * 8 bytes
Kernel Size	3	3	3
Stride	2	2	2

used parameters for each single layer:

Table 4-2 Alex-Net Pooling Layers parameters

4.4. Local Response Normalization layer (LRN)

Recall from Section 3.2.5 equation (1) that shows the normalization functionality; normalizing around the local neighborhood of the excited neuron and make it even more sensitive as compared to its neighbors. What is going to be done now, introducing the hardware blocks that would calculate this equation.

4.4.1. Computing engines

Each process in equation (1) will be defined by hardware block.

Summation of squares

That could be done by a tree of adders that adds the input squares as shown in Figure 4.8. Recall that the sum runs over n (n=5) adjacent kernel maps at the same spatial position, so 3 adders only are needed and will be reused. Finally, multiply by α and add k to get the denominator result of equation (1).

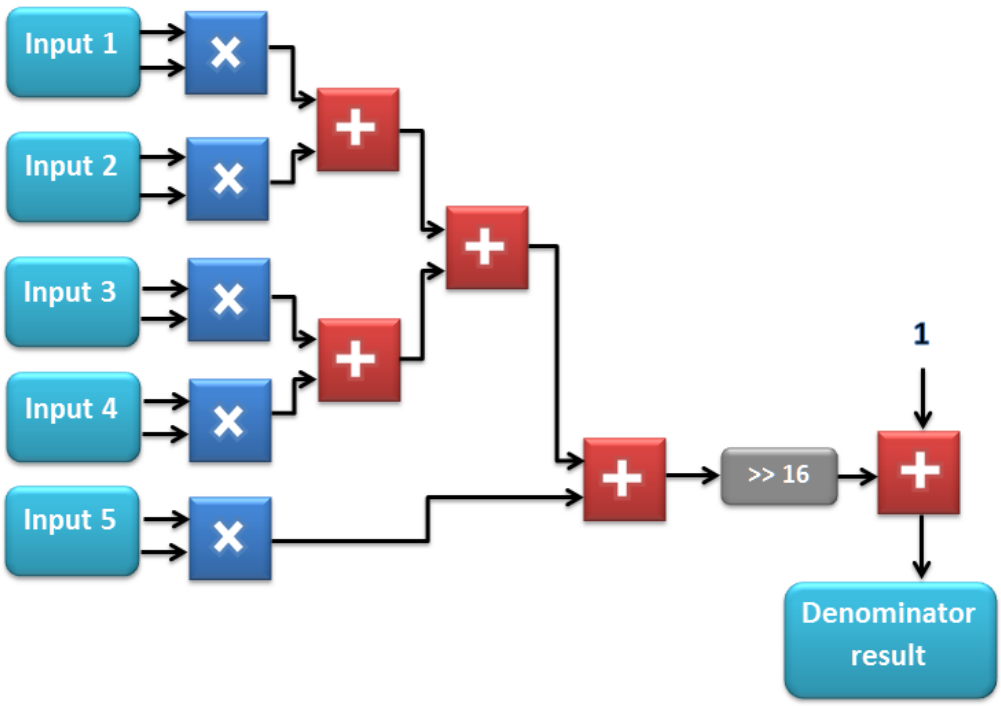


Figure 4.8 Norm Square and tree adder engine

Fixed point division

The idea is to use 2 integer divisions as illustrated in; the first one calculates the integer part and the second calculates the fraction part, then both parts concatenated to obtain the final result. Dividend and divider are fixed point numbers with; (1) S is 1 bit represents the sign, (2) I bits represents the integer part and (3) F bits represents the fraction number, the fixed point division result is also representing in the same format. In this network divider always larger than one, so the fixed point division is designed to work properly only when the divider is larger than one and any other values won't act properly. Sign bit is considered but actually no need for it because the Pool and RELU output is always positive.

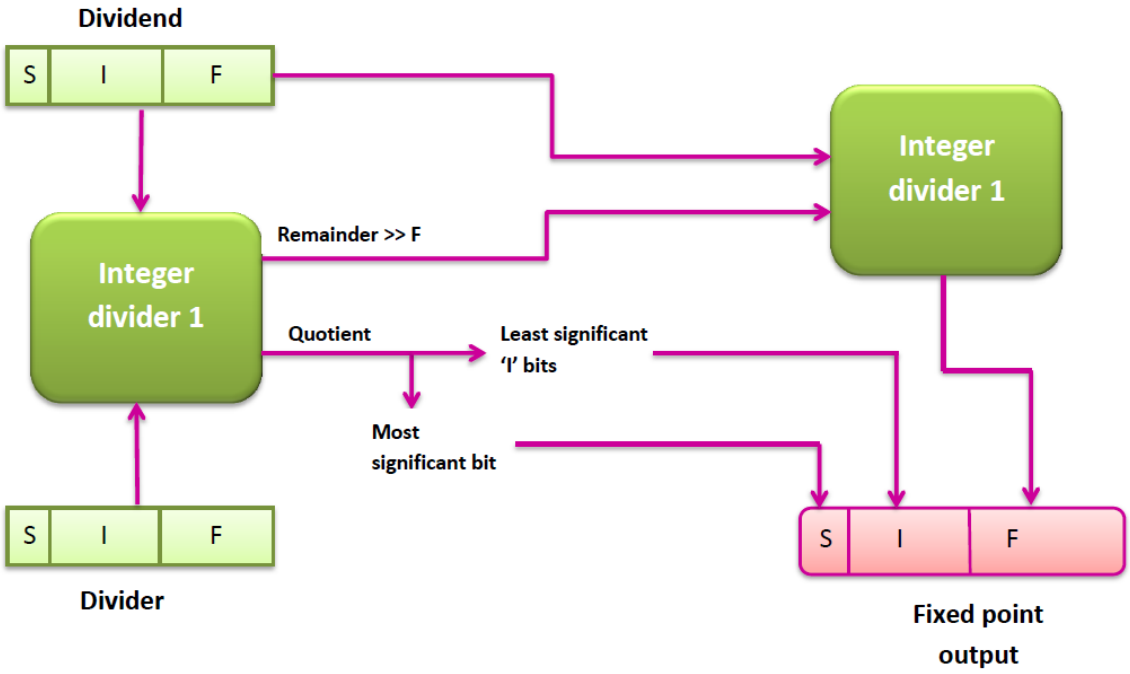


Figure 4.9 Division engine in Norm layer

4.4.2. Controlling structure

Figure 4.10 illustrates the controlling structure; (1) multiple input caches are needed for LRN as it needs to get the whole depth in one buffer to be able to apply squaring process and the tree of adders, so number of input caches in this design is equal to number of elements of the depth of the previous Pool layer to the LRN, (2) addressing unit is needed to access the caches parallel and it's controlled by the control unit using

the start Norm signal, (3) transition module is needed to get the n (recall n=5) elements of the summation by calculating the start and end position of the summation, (4) after computing each output element store it in one cache if the following layer is group 1 Conv (like Conv 2) or 2 caches if the following layer is group 2 Conv (like Conv1).

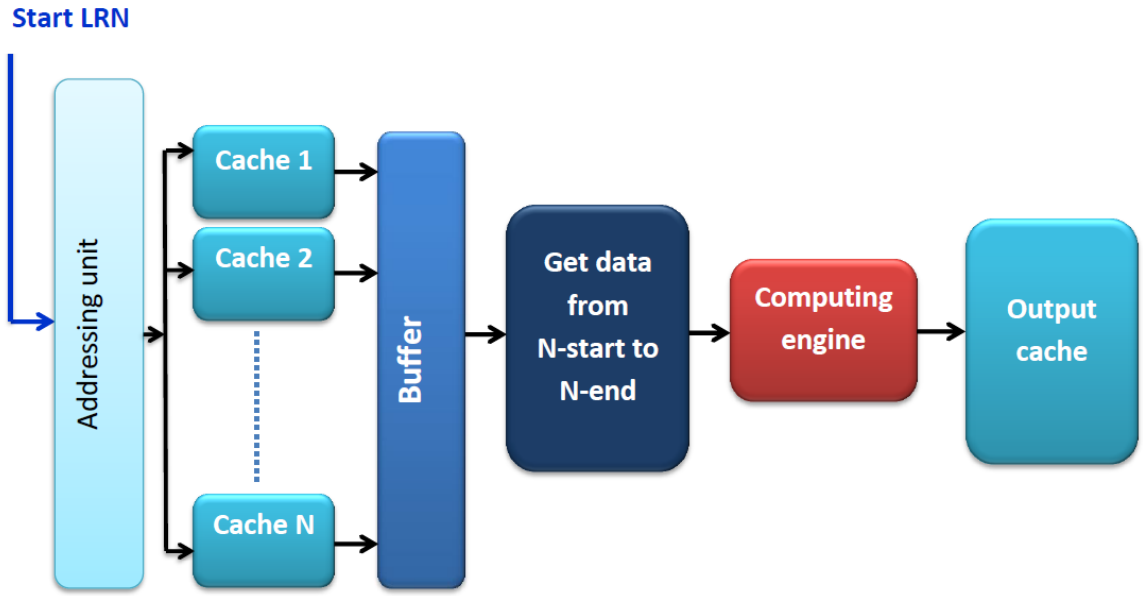


Figure 4.10 Control structure for Norm layer

4.5. Fully Connected Layer

After discussing the main three layers that are repeated in the first five super layers, the last layer is fully connected layer. Figure 4.11 shows the basic building block of the fully connected layer which is similar to that of the convolutional layer .It mainly depends on multiplying the input with the corresponding weight in the weight matrix and accumulates the result in the register.

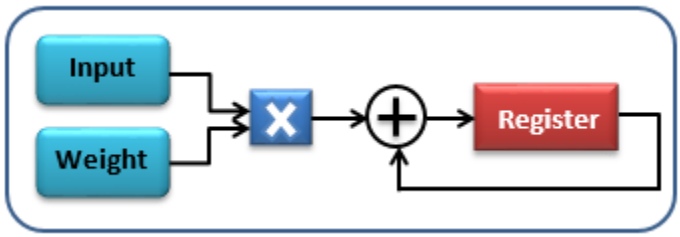


Figure 4.11 FC engine

The main operation of the Fully Connected layer as described before is multiplying a 2-Dimensional matrix of weights with an array of inputs so each output from the fully connected layer is due to the multiplying of a row from the weights' matrix with the inputs vector as illustrated in the Figure 4.12 .

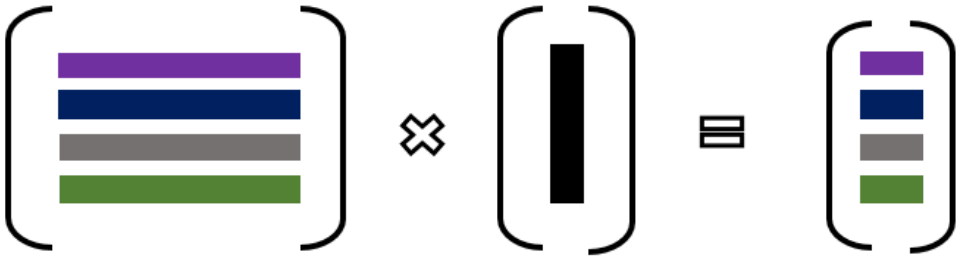


Figure 4.12 FC operation

The main target of the project is acceleration of a network so in this layer uses 2 techniques to speed up its operations:

1. Parallelism

The main idea of parallelism in this layer is using parallel engines of the building block corresponding to the number of rows of the weights` matrix but due to the large number of rows only part of the rows is taken in parallel and after getting the outputs corresponding to these rows another rows are taken as illustrated in Figure 4.13 .

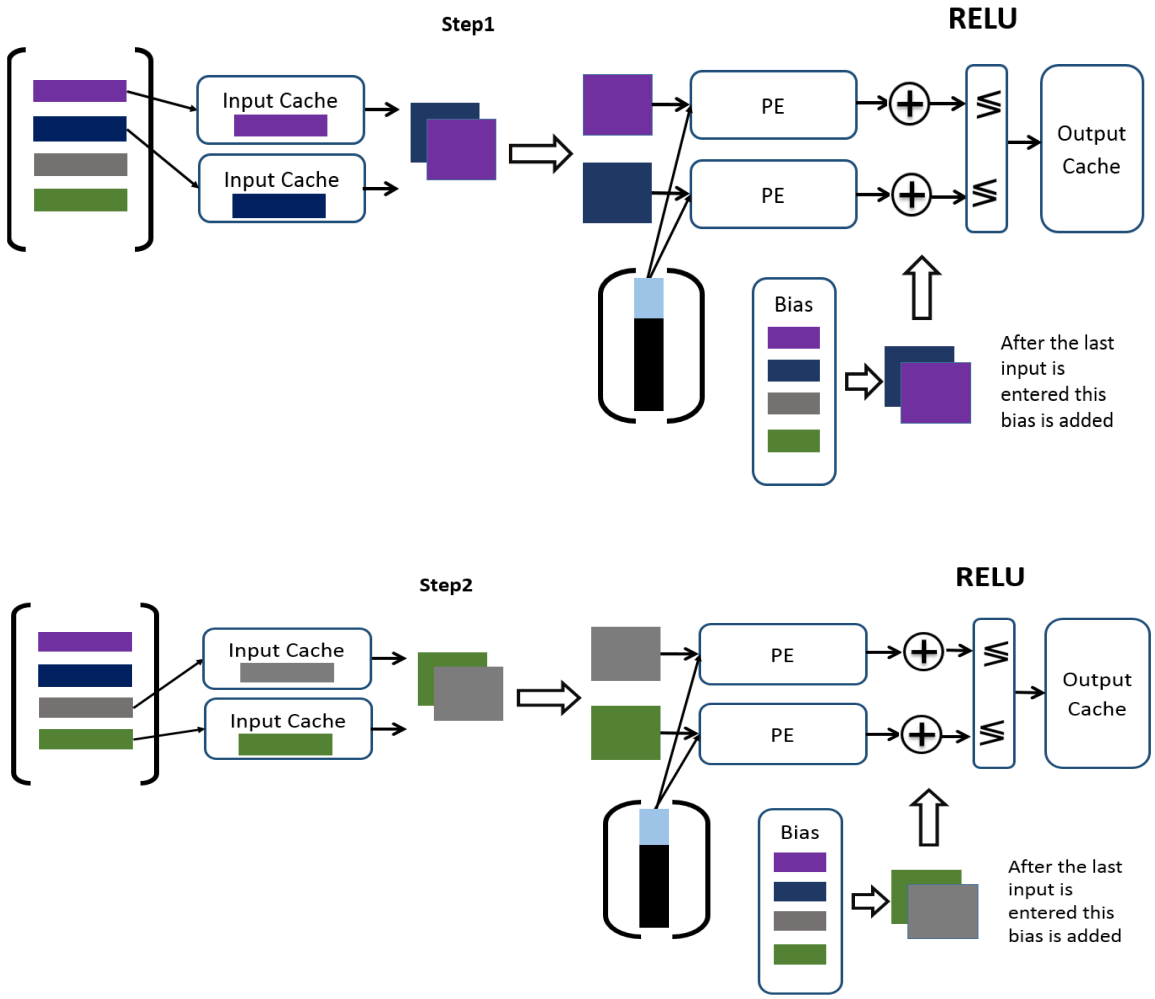


Figure 4.13 FC parallelism

2. Pipelining

For more speeding up, pipelining is used. In the layer`s design there is a cache for each PE at its weight port and only one cache for all the PEs at the output. So pipelining occurs in the weights' cache as illustrated in Figure 4.14 .

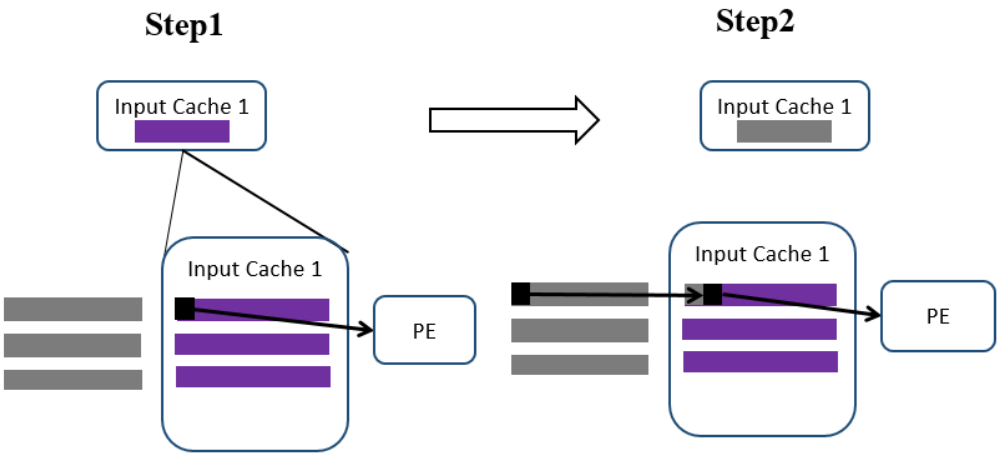


Figure 4.14 pipelining inside FC caches

In Alex-Net, there exists three fully connected layers so in the following table here are the used parameters for each single layer:

Table 4-3 Alex-Net FC Layers parameters

layers	FC6	FC7	FC8
Number of Parallel Engines	64	64	25
Input Size of each Parallel Engine	9216	4096	4096
Output Size of each Parallel Engine	4096	4096	1000
Cache size Before Parallel Engine	16K * 2 bytes	4K * 2 bytes	4K * 2 bytes
Number of serial loading of new rows to the PE	64	64	40

For different design using different number of PEs can be given as:

Number of serial loading of new rows to the PE =

$$\frac{\text{Output Size of each Parallel Engine}}{\text{Number of Parallel Engines}}$$

Number of cycles for executing any fully connected layer =

Number of serial loading of new rows to the PE x Input Size of each Parallel Engine.

4.6. Reshape function

The MATLAB function Reshape is used as a transition between the Pooling 5 layer and the first fully connected layer(FC6) to convert the 3D matrix output from the operations from Conv1 layer to pooling 5 layers to 1D column matrix to be multiplied by the fully connected layer weights.

- Input matrix dimensions= 6*6*256.
- Output matrix dimensions= 9216*1.

Hardware implementation:

The memories used to store the output of pooling 5 layer is implemented as cache array, the reshape output is stored in a cache of size 216 *16 bits. The main idea is address control, to output the whole depth once and store each location in the desired position. Example: location 0 from the output caches, gives 256 value, value 0 is stored in address 0 in the reshaped caches, while value 1 is stored in address 36 and so on. Location 6(1st column in each depth), gives 256 value, value 0 is stored in address 1, value 2 is stored in address 37. Finally, the output cache is reshaped to fit the next fully connected layers.

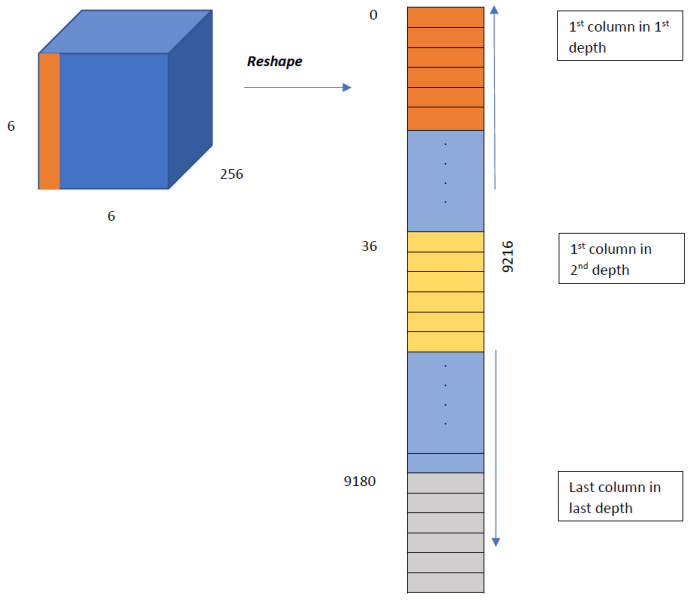


Figure 4.15 Reshape operation

4.7. Summary

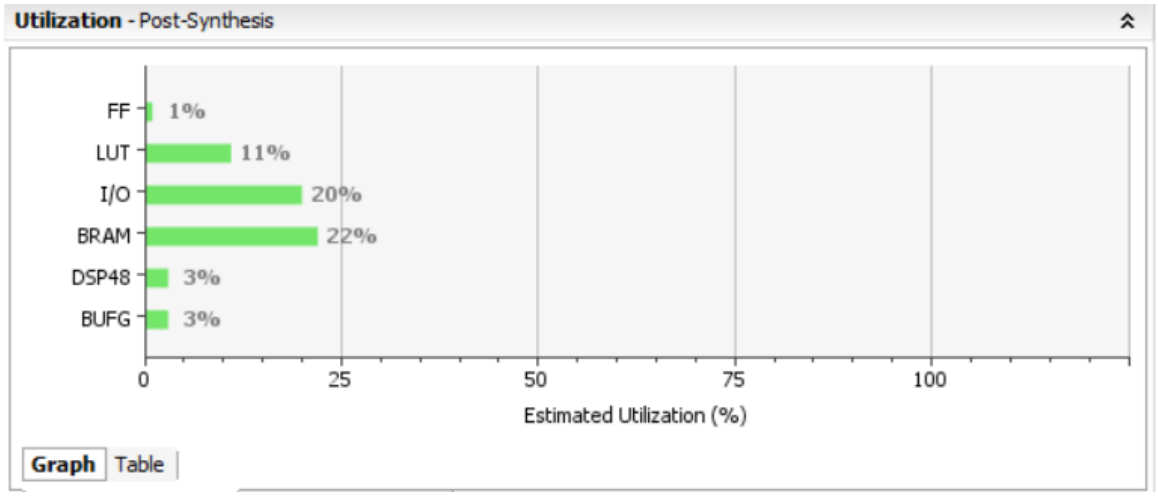
This chapter provides a discussion on the project's chosen design with the details of how each layer of the 5 main layers of Alex-Net is implemented, and accelerated using two main techniques, parallelism of resources and pipelining inside some layers, tabulates the different layers of the same type to show the slight difference between the implementation of each one and the other.

Chapter 5.
Synthesis and implementation

In this chapter the synthesis and implementation results are shown to demonstrate the FPGA used resources, timing and power.

5.1. Synthesis

After synthesizing the design on FPGA Virtex 7 using the VIVADO synthesis tool, Figure 5.1shows the utilization of the resources.



Utilization - Post-Synthesis

Resource	Estimation	Available	Utilization %
FF	10324	866400	1.19
LUT	47358	433200	10.93
I/O	168	850	19.76
BRAM	320	1470	21.77
DSP48	107	3600	2.97
BUFG	1	32	3.12

Graph **Table**

Figure 5.1 Post synthesis utilization summary

Figure 5.2 shows how each module is synthesized and the number of its resources.

Name	Slice LUTs (433200)	Slice Registers (866400)	F7 Muxes (216600)	F8 Muxes (108300)	Block RAM Tile (1470)	DSPs (3600)	Bonded IOB (850)	BUFGCTRL (32)
top_test	47358	10324	160	2	320	107	168	1
add_pool_1 (POOL_address...	295	154	0	0	0	2	0	0
bias1 (BiaseRom)	369	128	0	0	0	0	0	0
biasAdder (addBias)	0	1536	0	0	0	0	0	0
cache_after_Pool (RAM_AR...	0	0	0	0	48	0	0	0
conv1 (conv)	6144	3072	0	0	0	96	0	0
CU (controlUnit)	11203	534	0	0	0	2	0	0
dataPixels (input_memo)	0	0	0	0	32	0	0	0
Naddresses (norm1_addres...	15620	97	0	0	0	0	0	0
norm_1 (norm1)	251	0	0	0	0	7	0	0
norm_store (norm_storage)	257	131	0	0	0	0	0	0
Output (RAM_ARR__param...	3899	0	0	0	192	0	0	0
pool1 (POOL)	3828	3072	160	2	0	0	0	0
tr (transition_1)	769	20	0	0	0	0	0	0
WA (RAM_ARR)	1536	0	0	0	48	0	0	0

Figure 5.2 Synthesis utilization report

Analyzing the previous results of the main modules:

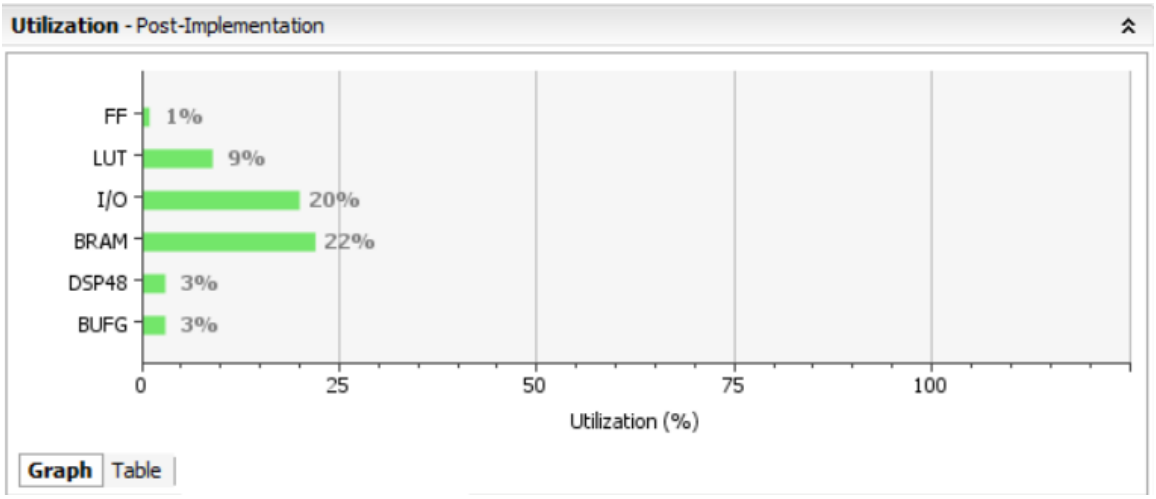
- Module 1(Data input memory) → **32 BRAM** each of size 36Kbit.

```
ROM:
+------------+------------+---------------+----------------+
|Module Name | RTL Object | Depth x Width | Implemented As |
+------------+------------+---------------+----------------+
|input_memo  | extrom     | 65536x16      | Block RAM      |
+------------+------------+---------------+----------------+
```

- Module 2(weight array) → **48 BRAM** each of size 18 Kbit
  The weight array is composed of 96 weight cache each occupy 0.5 BRAM, and **1536 LUTs** for the 16*96 output wires.
- Module 3(Convolution 1) → **96 DSPs** for the 96 parallel engines used in the MAC operations, **3072 slice registers** one for each output.
- Module 4 (bias adder) → **1536 slice registers** for the 16*96 outputs.
- Module 5 (Output Cache array) → **192 BRAMS** each of size 36Kbit where each cache utilizes 2 BRAMS.
- Module 6 (Pool 1) → **3072 slice registers** one for each output having 16*96 Inputs and outputs.
- Module 6(Cache after Pool array) → the same as weight array.
- Module7 (Norm1) → **7 DSPs** used for the squaring of the elements corresponding to the equation.

5.2. Implementation

After implementing the design on the previous mentioned Virtex 7 FPGA using the VIVADO implementation tool, Figure 5.3 shows the utilization of the resources.



Utilization - Post-Implementation

Resource	Utilization	Available	Utilization %
FF	10342	866400	1.19
LUT	38975	433200	9.00
I/O	168	850	19.76
BRAM	320	1470	21.77
DSP48	107	3600	2.97
BUFG	1	32	3.12

Graph **Table**

Figure 5.3 Post implementation utilization summary

The following figure shows how each module is placed and routed on Virtex 7 FPGA, and the number of its resources.

Name	Slice LUTs (433200)	Slice Registers (866400)	F7 Muxes (216600)	F8 Muxes (108300)	Slice (108300)	LUT as Logic (433200)	LUT Flip Flop Pairs (433200)	Block RAM Tile (1470)	DSPs (3600)	Bonded IOB (850)	BUFGCTRL (32)
top_test	38975	10342	160	2	11659	38975	40359	320	107	168	1
add_pool_1 (POOL_address...	294	154	0	0	88	294	292	0	2	0	0
bias1 (BiaseRom)	338	128	0	0	271	338	367	0	0	0	0
biasAdder (addBias)	0	1536	0	0	763	0	1536	0	0	0	0
cache_after_Pool (RAM_AR...	0	0	0	0	0	0	0	48	0	0	0
conv1 (conv)	5210	3072	0	0	1772	5210	5491	0	96	0	0
CU (controlUnit)	5039	534	0	0	3802	5039	5240	0	2	0	0
dataPixels (input_memo)	32	3	0	0	31	32	34	32	0	0	0
Naddresses (norm1_addres...	15551	97	0	0	4539	15551	15595	0	0	0	0
norm_1 (norm1)	249	0	0	0	101	249	249	0	7	0	0
norm_store (norm_storage)	257	131	0	0	96	257	284	0	0	0	0
Output (RAM_ARR__param...	4087	0	0	0	1661	4087	3554	192	0	0	0
pool1 (POOL)	3812	3072	160	2	2288	3812	5251	0	0	0	0
tr (transition_1)	760	20	0	0	334	760	768	0	0	0	0
WA (RAM_ARR)	1363	15	0	0	523	1363	1349	48	0	0	0

Figure 5.4 Implementation utilization report

Analyzing the previous results, same as the utilization report of the synthesized design but less resources specially LUTs due to optimization option.

Timing analysis results:

The following figure shows the setup and hold time slack, which are equal to zero which means the timing constraints are met at clock frequency 100 MHz

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.146 ns	Worst Hold Slack (WHS):	0.062 ns	Worst Pulse Width Slack (WPWS):	4.600 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:	28858	Total Number of Endpoints:	28858	Total Number of Endpoints:	11123

All user specified timing constraints are met.

Figure 5.5 Design timing summary

Power analysis:

The following two figure shows the power consumption of the super layer1 on the Virtex 7 FPGA, total power on chip = 1.141 watts.

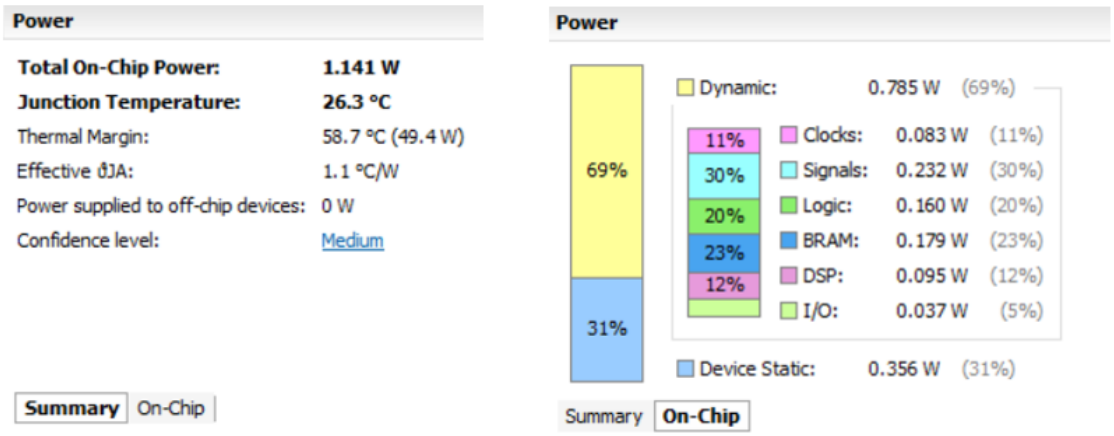


Figure 5.6 Power report summary

5.3. Summary

This chapter provides the results of the synthesis and implementation of super layer1 in Alex-Net, as the kit can`t fit all the design because it`s a first trial design which would be modified afterwards, using Vivado. Also it shows the number of resources utilized on the chosen kit, which is Virtex 7, the timing constraints and the power consumption of the design.

Chapter 6.
Optimization

In order to minimize the hardware needed across the network, the power consumption and minimize the time required for the classifications, optimization techniques as pipeline between network stages, replacing DSP multiplier by power efficient multiplier were used to achieve better performance from aspect of time, power and area. This section discuss briefly the optimization technique used and its algorithms, then shows the area, power and time reduction achieved by applying these techniques.

6.1. Pipeline approach

As the FPGA implements the logic for all the operations even if it doesn't work simultaneously. The approach is to rearrange the operations into a sequence (rearrange the algorithm into a pipeline), where each stage can operate simultaneously with the other stages. Pipelining tends to be faster than the state machine approach for accomplishing the same algorithm and it can even be more resource efficient [30].

The difficult part of a digital logic pipeline is that the pipeline runs and produces outputs even when the inputs to the pipeline are not valid as shown in Figure 6.1. Therefore, the algorithm must handle the signaling associated with pipeline logic.

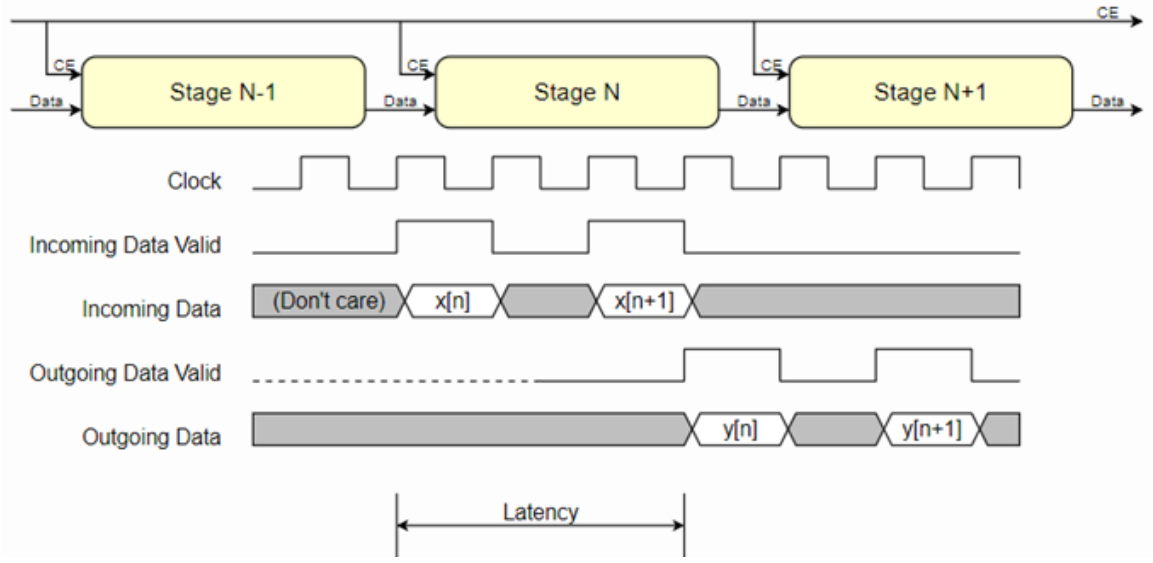


Figure 6.1 Data flow between pipeline stages

6.2. Pipeline between the convolution and pooling stages

Applying the pipeline approach between the convolution and pooling stages as they are the first two stages in the algorithm. The rest of this section considers Conv1 and Pool1 in Alex-Net as an illustrative example for simplicity.

Typically, in Alex-Net, Conv1 and Pool1 parameters are (kernel size=11x11x3, output size = 55x55x96), (kernel size=3x3, stride=2, output size=27x27) respectively as in Table 6-1. Which implies that, for the first output row of the pooling stage can be executed after the first three rows in Conv ‘s output is completed. For the second row of the pooling can be executed after the fourth and fifth rows of Conv are completed and so on as shown in Figure 6.2.

Figure 6.3 shows the replacement in caches for the Conv output to reduce the Conv output cache size, as the Conv outputs are not needed after the pooling is done.

- As the algorithm used in the design apply the convolution sequentially across the filter, each output in Conv1 require 363 cycle (11x11x3), similarly for the pooling each output needs 9 cycles (3x3), to produce a complete output row of pooling1 it takes (27x9= 243 cycle) which less than the number of cycles needed for one output of the convolution layer, hence pooling1 can operates during the convolution output is computed.

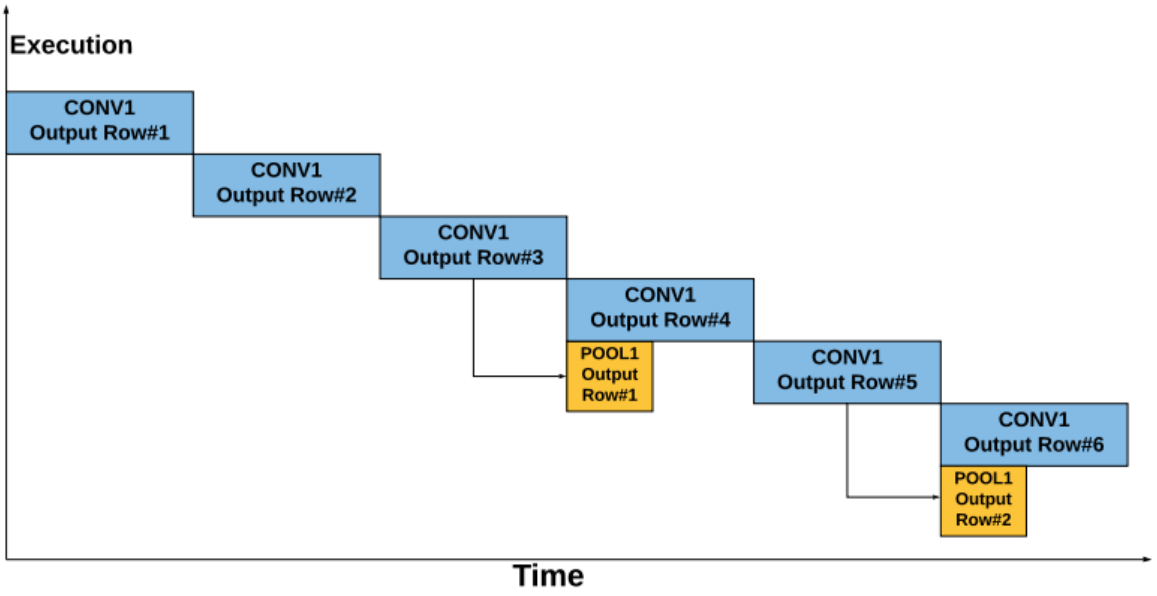


Figure 6.2 Execution of convolution and pooling stages in pipeline

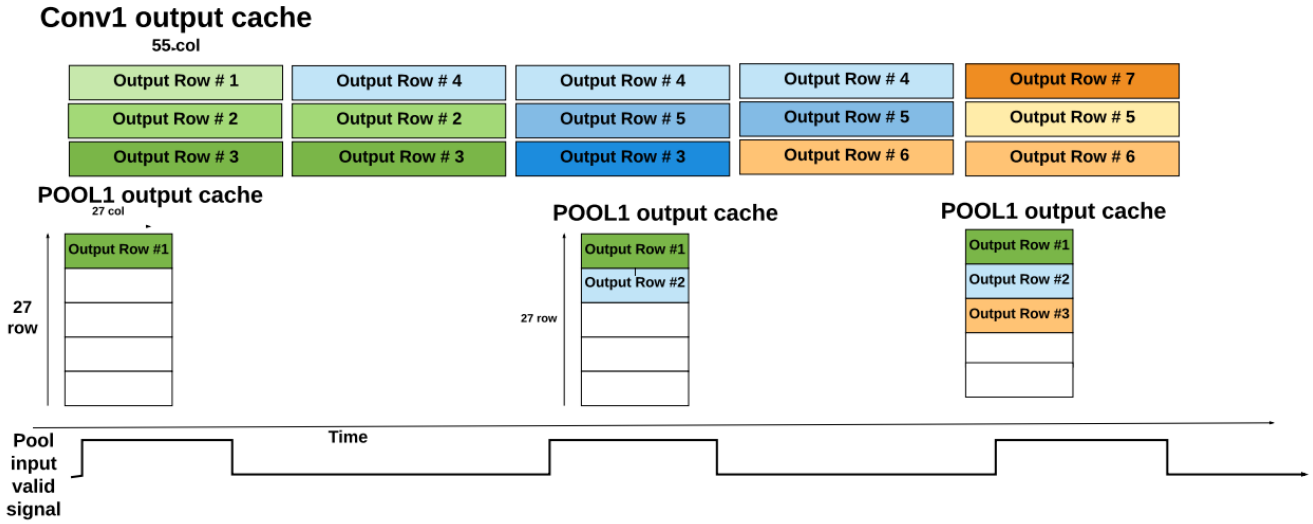


Figure 6.3 Replacement in Conv1 output cache

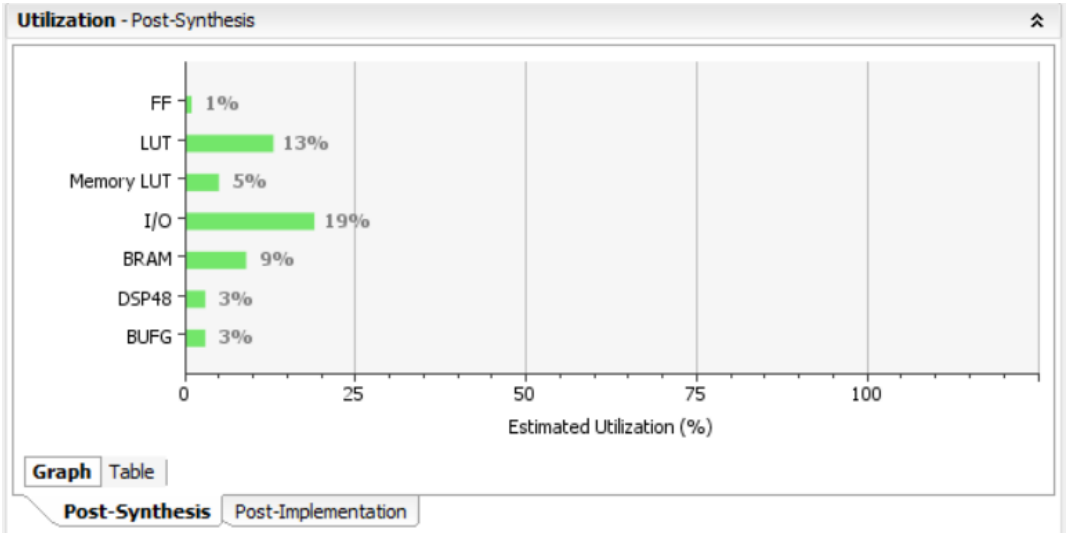
Table 6-1 Layer 1 parameters

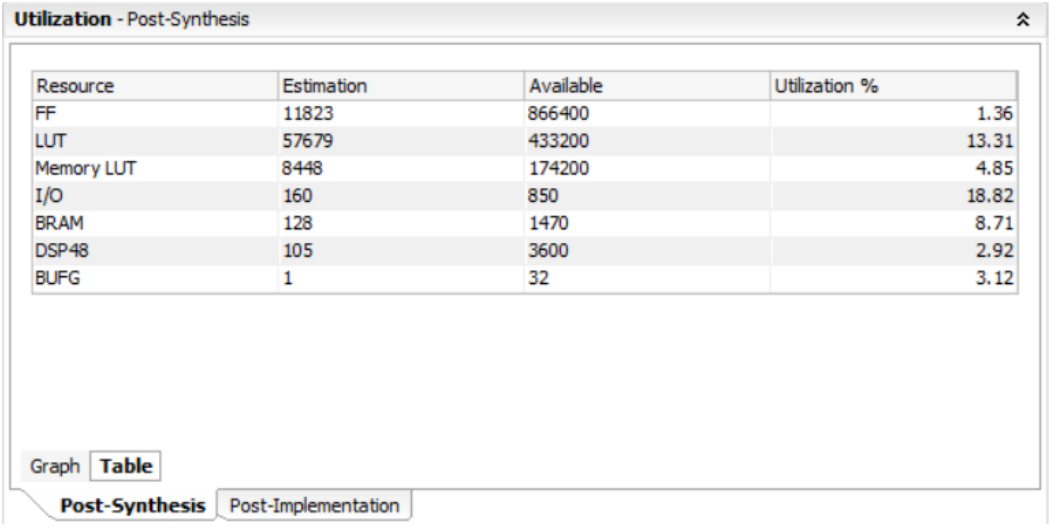
layer	Conv1	Pool1
Input size	**227x227x3**	**55x55x96**
Output size	**55x55x96**	**27x27x96**
Channels	**96**	**96**
Filter Size	**11x11x3**	**3x3**
Stride	**4**	**2**
Number of cycles for single output	**11x11x3 =363**	**3x3=9**
Number of cycles for row output	**55x363= 19965**	**27x9=243**
Latency	**0**	**55x3x363 for the first output row** **55x2x363 for the remaining output rows**

6.3. Pipelined design synthesis

Super layer 1 (Convolution 1, pooling 1, normalization 1) is synthesized and implemented to verify that the pipelined approach reduces the needed resources and also reduces the power and time needed to execute the layers' operations.

The design is synthesized on the previous mentioned Virtex 7 FPGA using the VIVADO synthesis tool. Figure 6.4 shows the percentage of the utilized resources from the total FPGA resources.





Resource	Estimation	Available	Utilization %
FF	11823	866400	1.36
LUT	57679	433200	13.31
Memory LUT	8448	174200	4.85
I/O	160	850	18.82
BRAM	128	1470	8.71
DSP48	105	3600	2.92
BUFG	1	32	3.12

Figure 6.4 Post synthesis utilization summary

The previous figure shows that the utilization of resources is reduced compared to the original design especially in the BRAMs utilization which decreased from 22% to 9% which a significant reduction is saving the area and power as will be shown in the design implementation results.

Figure 6.5 shows the utilization report of the pipelined design, declaring how each module is synthesized and number of resources used by each module.

Name	Slice LUTs (433200)	Slice Registers (866400)	F7 Muxes (216600)	F8 Muxes (108300)	Block RAM Tile (1470)	DSPs (3600)	Bonded IOB (850)	BUFGCTRL (32)
top_test	57679	11823	160	2	128	105	160	1
add_pool_1 (POOL_address...	363	183	0	0	0	0	0	0
bias1 (BiaseRom)	369	128	0	0	0	0	0	0
biasAdder (addBias)	1536	1536	0	0	0	0	0	0
cache_after_Pool (RAM_AR...	3652	0	160	2	48	0	0	0
conv1 (conv)	6144	3072	0	0	0	96	0	0
Conv1RamOut (DP_RAM_A...	12384	1536	0	0	0	0	0	0
CU (controlUnit)	11419	468	0	0	0	2	0	0
dataPixels (input_memo)	0	0	0	0	32	0	0	0
Naddresses (norm1_addres...	15626	97	0	0	0	0	0	0
norm_1 (norm1)	251	0	0	0	0	7	0	0
norm_store (norm_storage)	257	131	0	0	0	0	0	0
pool1 (POOL)	0	3072	0	0	0	0	0	0
tr (transition_1)	769	20	0	0	0	0	0	0
WA (RAM_ARR)	1536	0	0	0	48	0	0	0

Figure 6.5 Implementation utilization report

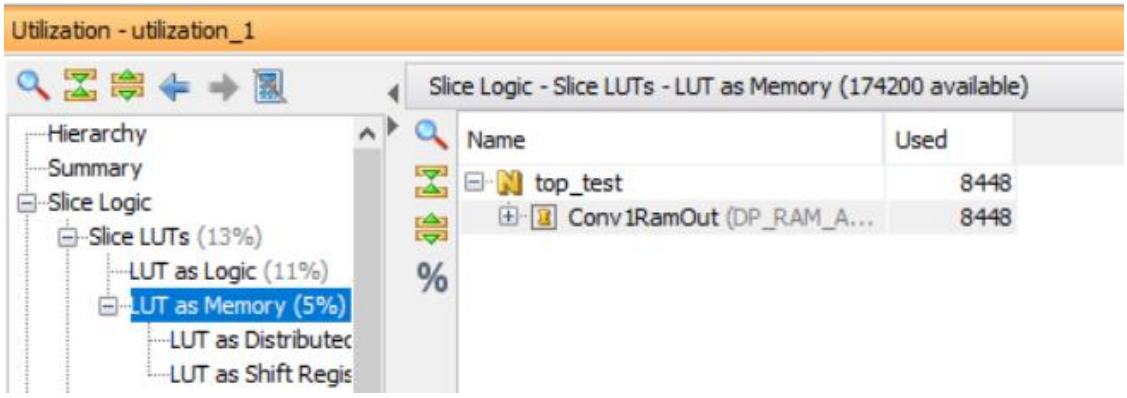


Figure 6.6 LUTs resources of output memory

Analyzing the previous results, all modules are synthesized as before in the original design. The noticeable difference is the utilization of the memory used to store the convolution output, in the original design all the output of the convolution layer is stored then the pooling operation starts, in the pipelined approach not all the outputs are stored, only the first 3 rows are stored in a dual port RAM and immediately the pooling operation

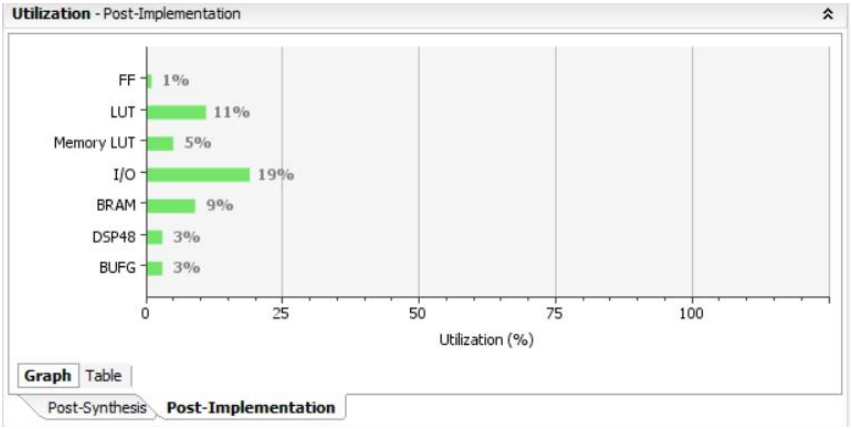
starts and when finished the dual port RAM is reused to store the next three rows and so on, so a large portion of the memory is saved.

Comparing between the memory resources utilization, the output cache in the original design which takes 192 BRAMs each of size 36Kbits, and in the pipelined design which takes 96*88= 8448 LUTs as memory shown in Figure 6.6, 88(28 * 16 bits) LUTs to save 3 rows for each filter output, the pipelined approach uses less memory resources.

6.4. Pipelined design implementation

The pipelined design is implemented on the Virtex 7 FPGA, to check the utilization after the place and route phase, power and timing summary.

Figure 6.7 shows the percentage of the utilized resources from the total FPGA resources after implementing the design.



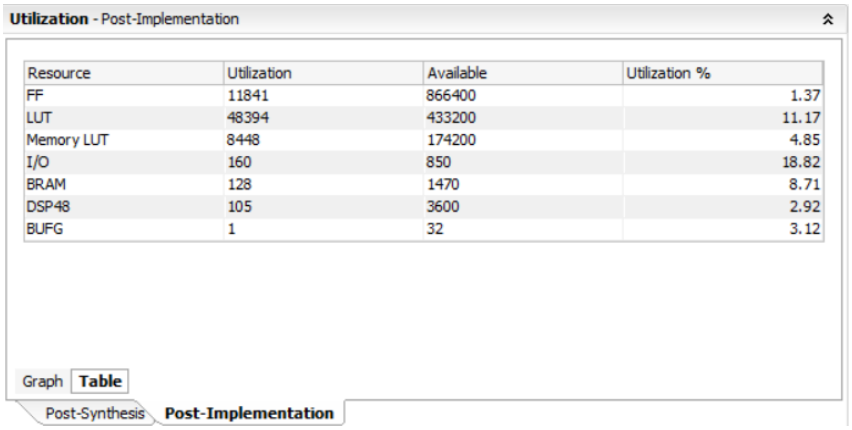


Figure 6.7 Post implementation utilization summary

Figure 6.8 shows the utilization report of the pipelined design, declaring how each module is implemented and number of resources used by each module.

Name	Slice LUTs (433200)	Slice Registers (866400)	F7 Muxes (216600)	F8 Muxes (108300)	Slice (108300)	LUT as Logic (433200)	LUT as Memory (174200)	LUT Flip Flop Pairs (433200)	Block RAM Tile (1470)	DSPs (3600)	Bonded IOB (850)	BUFGCTRL (32)
top_test	48394	11841	160	2	13636	39946	8448	49532	128	105	160	1
add_pool_1 (POOL_address...	356	183	0	0	114	356	0	354	0	0	0	0
bias1 (BiaseRom)	338	128	0	0	276	338	0	367	0	0	0	0
biasAdder (addBias)	1440	1536	0	0	1414	1440	0	2807	0	0	0	0
cache_after_Pool (RAM_AR...	3718	0	160	2	1640	3718	0	2791	48	0	0	0
conv1 (conv)	5183	3072	0	0	1780	5183	0	5337	0	96	0	0
Conv1RamOut (DP_RAM_A...	12219	1536	0	0	4163	3771	8448	13028	0	0	0	0
CU (controlUnit)	5235	468	0	0	3717	5235	0	5376	0	2	0	0
dataPixels (input_memo)	32	3	0	0	31	32	0	35	32	0	0	0
Naddresses (norm1_addres...	15557	97	0	0	4509	15557	0	15599	0	0	0	0
norm_1 (norm1)	248	0	0	0	108	248	0	248	0	7	0	0
norm_store (norm_storage)	257	131	0	0	91	257	0	283	0	0	0	0
pool1 (POOL)	0	3072	0	0	1022	0	0	2688	0	0	0	0
tr (transition_1)	758	20	0	0	321	758	0	762	0	0	0	0
WA (RAM_ARR)	1377	15	0	0	509	1377	0	1359	48	0	0	0

Figure 6.8 Implementation utilization report

Analyzing the previous results, same as the utilization report of the synthesized design but less resources specially LUTs due to optimization option.

Timing analysis results:

Figure 6.9 shows the setup and hold time slack, which are equal to zero which means the timing constraints are met at clock frequency 100 MHz

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.582 ns	Worst Hold Slack (WHS):	0.064 ns	Worst Pulse Width Slack (WPWS):	4.232 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:	91176	Total Number of Endpoints:	91176	Total Number of Endpoints:	20686

All user specified timing constraints are met.

Figure 6.9 Holding slack for pipelined design implementation

Power analysis:

Figure 6.10 shows the power consumption of the super layer1 on the Virtex 7 FPGA, total power on chip = 1.071 watts which is less than the power reported in the original design as the resources decrease specially the BRAMs which consumes 0.179 W in the original design, and 0.06 W in the pipelined design.

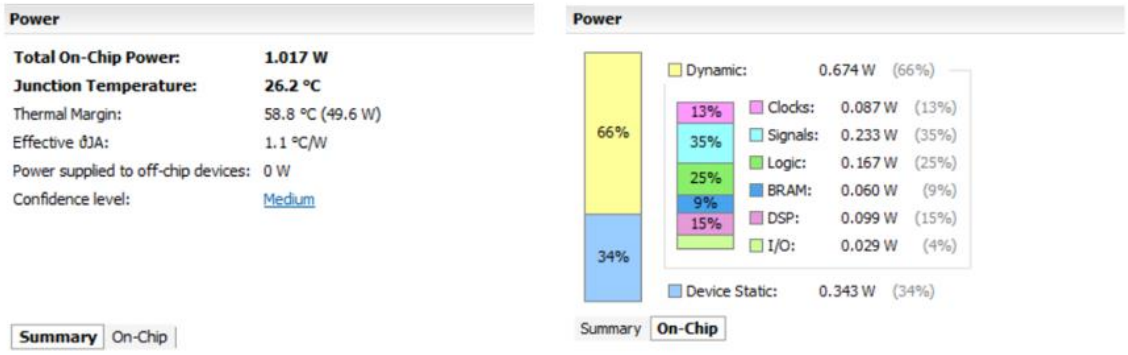


Figure 6.10 Power report summary

6.5. Summary

This chapter discusses a modification on the design discussed in chapter 4 based on pipelining, shows how this affects the number of resources and the simulation time emphasizing the target of the project which is accelerating Alex-Neton FPGA. Also it provides detailed results of synthesis and implementation of super layer 1 in Alex-Netafter this modification, to verify that the pipelined approach reduces the needed resources and also reduces the power and time needed to execute the layers' operations.

Chapter 7.
Results

7.1. Verification of RTL functionality

Back to the original purpose of the CNN is to predict the image and the object inside it, so the complete RTL network is tested by multiple images from the IMAGNET validation data set and compared to the MATLAB prediction.

Figure 7.1 shows a sample image from the tested images: which is harvester (reaper)

Figure 7.1 Sample image from dataset

Figure 7.2 shows the MATLAB results predicted correctly with maximum soft-max output 16.1635 and index predicted 596.

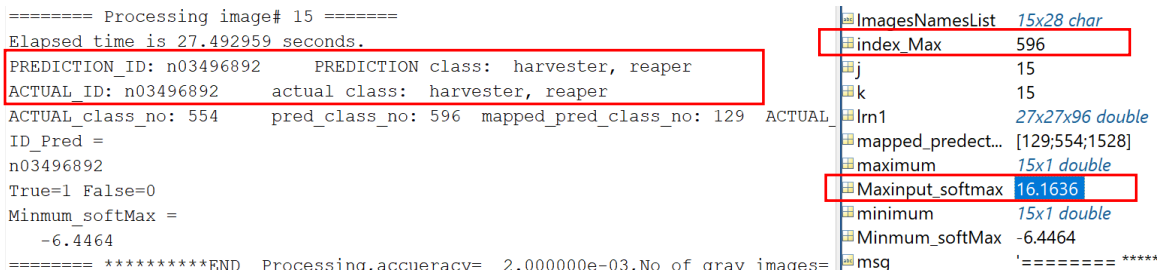


Figure 7.2 Software classification for input image

Figure 7.3 shows the RTL behavioral simulation results which is similar to MATLAB results with the output predicted correctly, index 595(starting from 0 index) and soft-max output 15.8125.

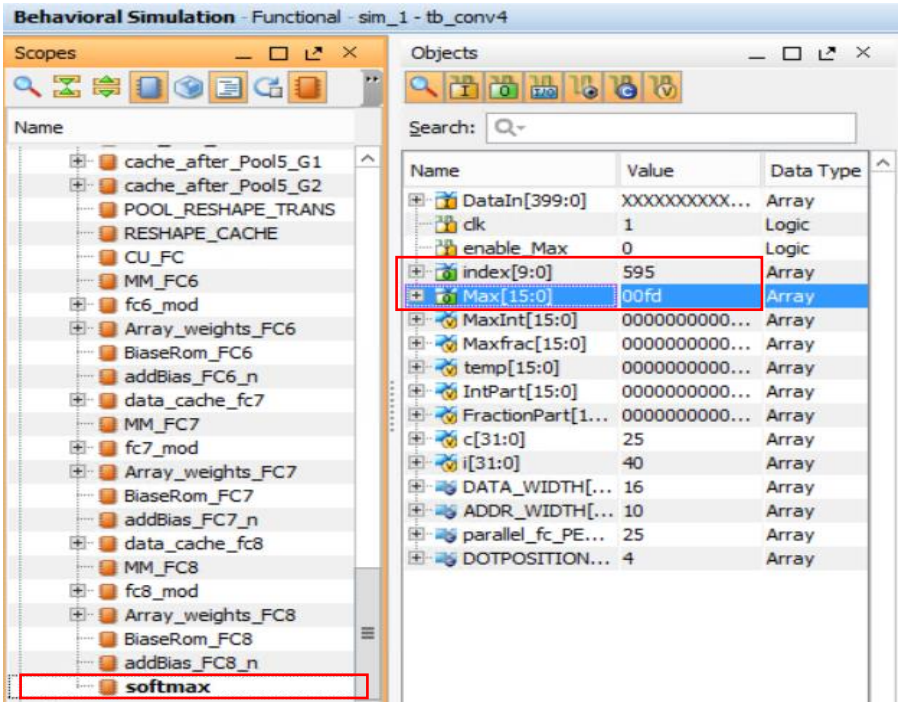


Figure 7.3 RTL classification for input image

Summarizing the comparison between the results of MATLAB and RTL simulation:

	MATLAB	RTL
MAX INDEX	596 (start from 1)	595 (start from 0)
SOFT MAX INDEX	16.16	15.8

7.2. Timing comparison between software and RTL

Table 7-1 shows the time taken by each layer to be executed by MATLAB (CPU) and on FPGA by the implemented design. It's clear that the bottleneck in simulation time was convolution layer, therefore it was the first design consideration to be accelerated and reduce it's time, the other layers were accelerated but the convolution has significant contribution in decreasing simulation time.

Table 7-1 Simulation time of Software and RTL

Layer	MATLAB simulation time (in S)	FPGA virtual simulation time
Conv1	7.8	11 ms
Pool1	0.37	65.6 µs
Norm1	1.83	700 µs
Conv2	7.9	8.7 ms
Pool2	0.3	1690 ns
Norm2	1	432.64 µs
Conv3	3.2	4 ms
Conv4	1.64	2.8 ms
Conv5	1.04	2.9 ms
Pool5	0.01	3240 ns

reshape	**0. 000018**	**3.4 µs**
Fc6	**0.37**	**6ms**
Fc7	**0.019**	**2.66 ms**
Fc8	**0.0074**	**1.68 ms**
Soft-max	**0.04**	**20 ns**
	25.5 s	**40.25 ms**

- Comparison with GPU is summarized in Table 7-2, the execution time depends on hardware resources available of the platform.

Table 7-2 Simulation time of Alex-Neton different GPUs

GPU	Forward (ms)
Proposed Architecture	**40.94**
GTX 1080 Ti	**4.31**
Pascal Titan X	**5.04**
Pascal Titan X	**5.32**
GTX 1080	**7.00**
Maxwell Titan X	**7.09**
GTX 1080	**7.35**

Maxwell Titan X	**7.55**

7.3. FPGA results

A sample from the accelerated design is implemented on the ZYNQ 702 board to check the operation on real time hardware. The chosen part is the first kernel from the 1st convolution layer and pooling operation on its output.

Figure 7.4 shows a sample of the pooling output from the real implemented design in the SDK XILINX tool.

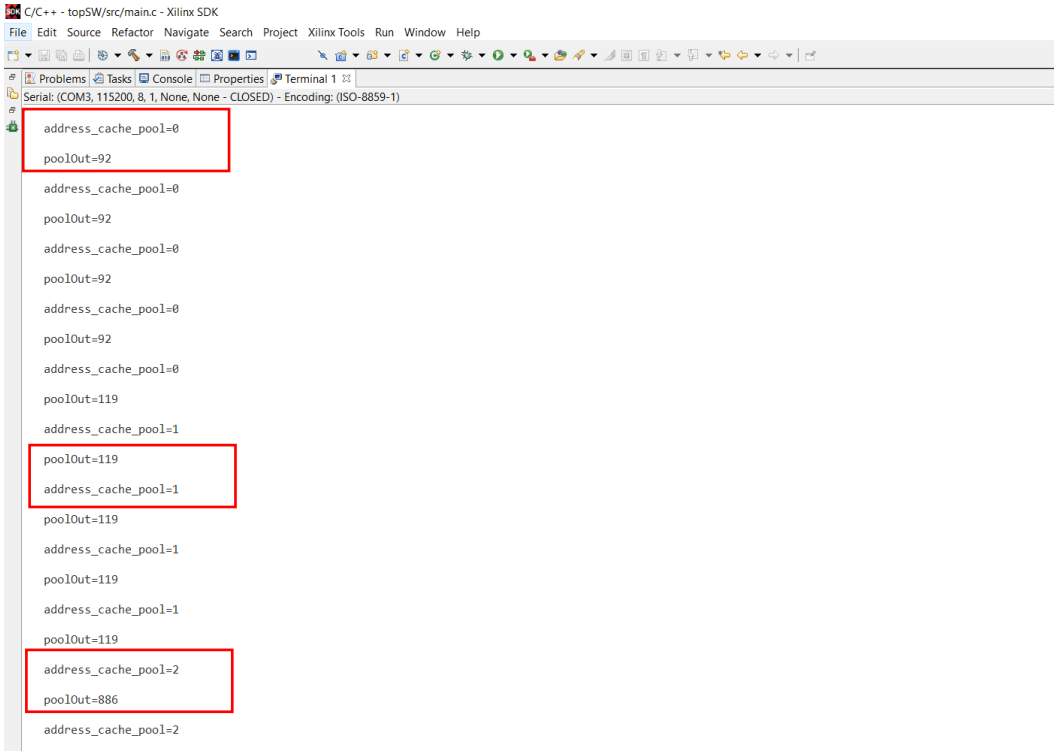


Figure 7.4 FPGA output for first super layer

Comparing the FPGA results with MATLAB results, same results are obtained. For example: the first location in SDK is 5c (hex) =5.75 and from MATLAB 5.746.

Therefore, the results in MATLAB, RTL, and FPGA are matched and the output is **predicted correctly.**

7.4. Summary

This chapter provides the results of the simulation of the whole design on Vivado and discusses the simulation timing results on Matlab and Virtual FPGA (Behavioral Simulation) comparing between them showing how the design is accelerated in RTL design compared to that running on software.

Chapter 8.
Conclusion and future work

8.1. Conclusion

In this thesis we demonstrate the acceleration of the forward path of a pre-trained Alex-Net on FPGA, introducing the parallelism and pipeline techniques that can be used to accelerate the network, and the hardware architecture used to implement it.

The proposed architecture was discussed in details. The parallelism and pipelining techniques to accelerate the network were illustrated clearly and their contribution to achieve better performance from area and power consumption point of view. When evaluating performance, it is clear that the GPU is currently most efficient in terms of execution time and throughput, though the FPGA is best in terms of power consumption and pipelining opportunity.

In closing, its believed that this work achieve the desired objective by showing that FPGAs can be used as a practical acceleration platform for deep convolutional neural networks. While, in its current state, this work does not convince deep learning practitioners to use FPGAs instead of GPUs, it's believed that the directions which make this reality have been identified.

8.2. Future work

8.2.1. Introduction

Future work concerns deeper analysis of particular mechanisms, new proposals to try different methods, or simply curiosity. This thesis has been mainly focused on parallel acceleration to speed up as possible without giving enough attention to (1) Pipelining on the Network Level, (2) using SD Card on FPGA, (3) Increasing the parallelism according to the FPGA available resources, (4) Improving the network accuracy using the pre-layer quantization, (5) Pruning the network to reduce the power and increase the throughput, (6) Computational skipping,(7)Introducing time sharing technique, and (8) Introducing PDR technique. And also, this thesis considers only one

picture to be processed which is saved in cache on board, so continuous input is not introduced. These points left for the future due to lack of time.

8.2.2. Pipelining on the Network Level

Applying pipeline in Conv1 & Pool1 lead to reduction in hardware utilization. Same approach can be applied across the network layers. The main idea is to pass the minimum data needed between layers to make the following layer start execution, so no layer has spare time as possible. It is expected to increase the overall network throughput and reduce the hardware resources significantly.

8.2.3. SD Card

Instead of initializing some ROMs in the design with the weights, bias values or the input image values which would take in some layers many number of code lines and the simulation tools won't be able to compile them, an alternative solution is using the SD card on the FPGA kit ,containing all the files of all the needed values, which would send the data before the start of the execution of the network operations to the targeted ROMs which would reduce the number of code lines but will take some time at first for data transfer due to its small bus width.

8.2.4. Increase the parallelism according to the FPGA available resources

As the network can be accelerated by increasing the number of parallel engine that can work simultaneously, the limitation on the available FPGA resources (Block RAMS, DSP slice, CLB). For example, considering Conv1 to illustrate: the filter size = 11x11x3, and there are 96 filters. Typically, the number of parallel engines used in current design = 96, each engine corresponds to one depth of output. Therefore, the convolution operation for one output take 11x11x3=363 cycle. To speed up the convolution operation we can use parallel multipliers in single depth i.e. 11x11 multiplier for each depth, hence the convolution output is done in 3 cycles.

8.2.5. Improve the network accuracy using the pre-layer quantization

As the accuracy was degraded after quantizing the network from (64-floating point) to (16-fixed point with 12 bit for integer and 4 bits' fractions for input data and 1 bit for integer and 15 bits' fractions for weights) which lead to decreasing in accuracy (from 53.4 to 52.7). The Pre-layer quantization can improve the network accuracy by quantizing each layer separately which lead to better results.

8.2.6. Pruning the network to reduce the power and increase the throughput

By eliminating all relatively small weights of neurons as they have insignificant contribution to the output. By applying pruning algorithms to rank the neurons in the network according to how much they contribute to the output, then remove the low ranking neurons from the network, resulting in a smaller and faster network therefore high throughput can be achieved.

8.2.7. Computational skipping

By skipping the operation on zero input in the network. As all the negative outputs are set to zero after RELU layers which generate sparsity in the network, therefore the operation on zero input can be skipped to reduce the power consumption across the network.

8.2.8. Time sharing

By using the same hardware for all the super layers (Conv– Pool– norm), and differentiate between their operation in time using different states for each layer.

8.2.9. PDR

By reconfiguring hardware for each layer, therefore can run all the network on one FPGA.

References

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

[2] K. He et al., "Deep Residual Learning for Image Recognition", 2016.

[3] "ImageNet Large Scale Visual Recognition Competition (ILSVRC)", Image-net.org, 2017. [Online]. Available: http://www.image-net.org/challenges/LSVRC/.

[4] K. Simonyan and A. Zisserman, "VERY DEEP ConvOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION", 2014.

[5] H. Li et al., "Acceleration of Deep Learning on FPGA", 2017.

[6] M. Peemen et al., "Memory-Centric Accelerator Design for Convolutional

[7] "Learning about deep learning", Recode, 2017. [Online]. Available: https://www.recode.net/2016/5/4/11634228/learning-about-deep-learning.

[8] "FPGA Acceleration of Convolutional Neural Networks - Nallatech", Nallatech, 2017. [Online]. Available: http://www.nallatech.com/fpga-acceleration-convolutional-neural-networks/.

[9] B. Moons et al., "Energy-Efficient ConvNets Through Approximate Computing", 2016.

[10] D. Patterson et al., Artificial neural networks. Singapore: Prentice Hall, 1996.

[11] Y. Qiao, J. Shen, T. Xiao, Q. Yang, M. Wen and C. Zhang, "FPGA-accelerated deep convolutional neural networks for high throughput and energy efficiency", Concurrency and Computation: Practice and Experience, vol. 29, no. 20, p. e3850, 2016.

[12] C. Zhang et al., "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks", 2015.

[13] "Deep Learning 101 - Part 1: History and Background", Beamandrew.github.io, 2018.[Online].Available:https://beamandrew.github.io/deeplearning/2017/02/23/deep_learning_101_part1.html.

[14] "alexnet_tugce_kyunghee.pdf - ImageNet Classification with Deep Convolutional Neural Networks Alex Krizhevsky Ilya Sutskever Geoffrey E Hinton Presented", Coursehero.com, 2018. [Online]. Available: https://www.coursehero.com/file/24481166/alexnet-tugce-kyungheepdf/. [Accessed: 11- Jul- 2018].Neural Networks?", 2017.

[15] T. ARTICLES, N. PRODUCTS, G. ELECTRONICS, C. PROJECTS, E. MICRO, V. Lectures, I. Webinars, I. Training, P. Search, T. DB, B. Tool, R. Designs and S. H.L., "Purpose and Internal Functionality of FPGA Look-Up Tables", Allaboutcircuits.com, 2018. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/purpose-and-internal-functionality-of-fpga-look-up-tables/. [Accessed: 11- Jul- 2018].

[16] Indico.desy.de, 2018. [Online]. Available: https://indico.desy.de/indico/event/7001/session/0/contribution/1/material/slides/0.pdf. [Accessed: 11- Jul- 2018].

[17] E. Nurvitadhi et al., "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?", 2017.

[18] V. Sze et al., "Efficient Processing of Deep Neural Networks: A Tutorial and Survey", 2017.

[19] Y. Chen, T. Krishna, J. Emer, and V. Sze, “Eyeriss : An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks Future of Deep Learning Recognit ion DCNN Accelerator is Crucial • High Throughput for Real-time,” IEEE Int. Solid-State Circuits Conf. , Feb. 2016.

[20] "An Intuitive Explanation of Convolutional Neural Networks", the data science blog, 2018. [Online]. Available: https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/. [Accessed: 10- Jun- 2018].

[21] "Understanding Convolutional Neural Networks for NLP", WildML, 2018. [Online]. Available: http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/. [Accessed: 10- Jun- 2018].

[22] "Activation Functions: Neural Networks – Towards Data Science", Towards Data Science, 2018. [Online]. Available: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6. [Accessed: 11- Jun- 2018].

[23] "CS231n Convolutional Neural Networks for Visual Recognition", Cs231n.github.io, 2018. [Online]. Available: http://cs231n.github.io/neural-networks-1/. [Accessed: 11- Jun- 2018].

[24] "A Quick Introduction to Neural Networks", the data science blog, 2018. [Online]. Available: https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/. [Accessed: 11- Jul- 2018].

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

[26] "Why dropouts prevent overfitting in Deep Neural Networks", Medium, 2018. [Online]. Available: https://medium.com/@vivek.yadav/why-dropouts-prevent-overfitting-in-deep-neural-networks-937e2543a701. [Accessed: 11- Jul- 2018].

[27] "Caffe | Deep Learning Framework", Caffe.berkeleyvision.org, 2018. [Online]. Available: http://caffe.berkeleyvision.org/. [Accessed: 11- Jul- 2018].

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

[29] "pmgysel/alexnet-forwardpath", GitHub, 2018. [Online]. Available: https://github.com/pmgysel/alexnet-forwardpath. [Accessed: 12- Jul- 2018].

[30] "Strategies for pipelining logic", Zipcpu.com, 2018. [Online]. Available: http://zipcpu.com/blog/2017/08/14/strategies-for-pipelining.html. [Accessed: 11- Jul- 2018].