A Highly Optimized Design Space Exploration Scheme for implementing Deep Convolution Neural Networks

Author:

M. SOHAIB UL HASSAN

00000171141

Supervisor:

DR. UMAR SHAHBAZ KHAN

Co-Supervisor:

DR SAJID GUL KHAWAJA

DEPARTMENT OF MECHATRONICS ENGINEERING

COLLEGE OF ELECTRICAL & MECHANICAL ENGINEERING

NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY

ISLAMABAD

MAY, 2020

A Highly Optimized Design Space Exploration Scheme for implementing Deep Convolution Neural Networks

Author

M. SOHAIB UL HASSAN

00000171141

MS-16

A thesis submitted in partial fulfillment of the requirements for the degree of

MS Mechatronics Engineering

Thesis Supervisor:

DR UMAR SHAHBAZ KHAN

Co-Supervisor:

DR SAJID GUL KHAWAJA

Thesis Supervisor's Signature: ____

DEPARTMENT OF MECHATRONICS ENGINEERING

COLLEGE OF ELECTRICAL & MECHANICAL ENGINEERING

NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY, ISLAMABAD

MAY, 2020

Declaration

I indorse that this research work titled "*A Highly Optimized Design Space Exploration Scheme for implementing Deep Convolution Neural Networks"* is my own work. The material used in this work from other sources has been properly referenced and it hasn't been presented elsewhere for assessment.

Signature of Student

M. Sohaib Ul Hassan

00000171141

Language Correctness Certificate

This thesis has been read by an English expert and is free of typing, syntax, semantic, grammatical and spelling mistakes. Thesis is also according to the format given by the university.

Signature of Student

M. Sohaib Ul Hassan

00000171141

Signature of Supervisor

Dr. Umar Shahbaz Khan

Copyright Statement

Dedicated to my exceptional family members and teachers

Abstract

Convolutional Neural Network (CNN) is an important machine learning algorithm. Due to its broad applications and classification accuracy it has become hot topic in recent times. Convolutional Neural Networks are both computationally expensive and have extensive memory accesses which has rendered it inefficient on general purpose computers. GPU implementations have improved the performance of algorithm but high energy consumption of GPUs doesn’t allow its usage in robotics and mobile embedded platforms. This study presents the implementation details of mapping Convolutional Neural Networks on field programmable gate arrays (FPGAs). Visual Geometric Group (VGG-16) Networks are the most admired CNN architectures in community. They have uniform and regular structure which is most suitable to be implemented on FPGA. So, a detailed discussion of mapping VGG-16 style networks on FPGA is presented. Flower Recognition example of Kaggle was used as case study. Training of a VGG style network was carried out on core i9 computer with NVIDIA GTX 1660 GPU. On dataset trained network achieved an accuracy of 90%. Trained CNNs are algorithmically simple to model and deploy. Xilinx Zynq Zedboard was used for analytical modeling and mapping of CNN. Trained CNN was partitioned into two parts hardware part and software part. Hardware part being comprised of computationally extensive convolutions and software part being comprised of computationally less expensive tasks such as Pooling layer, Fully Connected layer and SoftMax layer. Hardware part of CNN was mapped on Zynq-PL and software part was mapped on Zynq-PS. For different types of parallelism opportunities that exist in CNN workload, proposed methodology achieved inter output parallelism in design of hardware accelerator on Zynq-PL. Hardware design on Zynq-PL also took into consideration memory access patterns of convolution operation and optimized them to achieve good performance. For a complete network implementation, proposed methodology achieved a peak performance of 1.3 GMACCs at 120 MHz frequency and achieved a speed up of 4 times compared to software implementation on General Purpose Computer.

Key Words: FPGA, Convolutional Neural Network, VGG-16, Zedboard

Table of Contents

CHAPTER 1: INTRODUCTION

1.1 Motivation

Pattern recognition is quite a difficult thing for computers to accomplish. Tasks such as scene labeling, pattern recognition and image classification are very important in computer vision and robotics. Advancement in the field of computer vision has introduced new methods and techniques which are helpful in understanding objects in images. Previously recognizing objects consisted of techniques which required hand engineered features and hard coded algorithms. These techniques were found to be passive and not so impressive in efficiency. Advanced recognition methods consist of machine learning algorithms. These algorithms consist of models which are trained on images of a large dataset and once features in images are learnt, they can classify objects in an image. These learning models are usually based on high performance computing platforms.

Convolutional neural networks (CNNs) are one of the most impressive techniques which are currently being used in computer vision algorithms. These networks are sub class of Artificial Neural Networks (ANNs). Instead of hidden layers, CNNs comprise of convolutional layers, subsampling or pooling layers, activations layers which are usually sigmoid and fully connected layers (like ANN). The idea of Artificial Neural Network has been around for quite some time but only recently with the emergence of high-performance computers, it has been possible to train and deploy these CNN models. The progress in the last decade has been nothing short of amazing, currently CNNs rival human visual system in terms of recognition accuracy, for example FaceNet [1] has recognition accuracy of 96.7% which even exceeds human ability.

Ground breaking performance of these CNN models usually comes at the cost of huge amount of computation complexity. Operation of image recognition CNN usually requires billions and trillions of operations in one second. Add to that the preprocessing and postprocessing of images required to build a real time application, computation complexity becomes even greater. Graphical Processing Units (GPUs) were recently utilized to a greater effect to achieve high computational throughput required for these CNN models. Problem with use of these GPUs is that they consume a lot of power. GPU based implementation of these models in not possible in robotics system or mobile platforms. Robotics and mobile applications thus require embedding these CNN models on small, efficient and powerful computing platforms.

Different computing platforms have been explored recently like field programmable gate arrays (FPGAs) and Application Specific Integrated Circuits (ASICs). Each has its own pro and cons like ASICs implementation are very efficient solution in terms of power consumption but they are not so good for commercial applications due their large development cycle, huge NRE cost and non-configurability. FPGA implementation can be fast with low power consumption with a good programmability which is important for commercial applications.

1.2 Our Contribution

This study presents architecture scheme for highly efficient design space exploration of CNN on FPGA Platform. FPGA and its vendors have made frameworks which target the accelerating machine learning models. But, those solutions were not commercially available. So, we decide to build a proof of concept implementation of CNN on FPGA platform from scratch. Recent advances in both FPGA technology and machine learning algorithm has aggravated the problems in implementation of CNNs on FPGA. Increase in logic resources and external memory bandwidth on FPGA platforms makes the design space exploration large. On the other hand, deep learning algorithms keep getting bigger and complex each day. So, it is has become more difficult to implement deep learning algorithm on FPGA.

Main focus of this study is to build optimized accelerator for CNN on FPGA platform keeping in mind the architecture of the network and memory access optimization. Second part is key to achieving good performance and throughput. Many of the previous implementations have failed to address this aspect. Extensive experimentation showed that accessing memory to read weights and neurons is as important as speeding up the process itself. Both of these problems have dependency on each other. So, a careful realization of each part is needed to make an accelerator work. Previously most CNN implementations on FPGA have been of two type Systolic Architecture and Hardware Implementation. Most complete network implementations have been carried out using systolic architecture. Most frequently used network implementation when speeding up the process of convolution only is Hardware Implementation. This study presents a scheme for mapping CNN on FPGA which is mixture of both these techniques. Some of the key features of our work are listed below.

- Use of Zynq-7 ZC7020 Evaluation Board for implementation
- Implementation of VGG-16 style network for classification flowers of five different categories
- Analytical Modeling of the network in terms of Zynq implementation
- Image Acquisition and Preprocessing and subsampling tasks are performed on Zynq PS
- Convolution is performed on Zynq PL
- Tiled data transfer between PS and PL
- External memory access optimization by performing loop transformations depending upon dependency analysis of different loop iterations

CHAPTER 2: CONVOLUTIONAL NEURAL NETWORKS

This chapter presents a background of the Convolutional Neural Networks (CNNs), their origin and different types of CNNs that are considered state of art to solve real life problems. Section 2.1 discusses the basis of Neural Networks and also CNN is explained as a special class of Neural Networks. Section 2.2 gives introduction to different platforms which are currently being used to train Neural Network Models. Section 2.3 demonstrates different network architectures that are currently being used.

2.1 Convolutional Neural Network

This section presents the study of evolution of Neural Networks. Besides, General overview of mathematical representation of Neural Networks is discussed. Evolution and modeling of convolutional neural networks is provided in details. Convolutional Neural Network is a special class of Artificial Neural Network (ANNs). ANNs can be used in many pattern recognition problems like sound signals and real-life problems, whereas CNNs are specially used in pattern recognition. First, the working of Artificial Neural Networks is presented including. Secondly discussion on architecture and mathematical representation of Neural Network along with different network architectures is presented.

2.1.1 Introduction to Neural Networks

Artificial Neural Networks (ANNs) are hugely computational algorithms which have biological inspiration in their making. A Human brain usually consists of 86 billion neurons which are connected by approximately 10^{14} synapses where, each neuron takes its inputs at the dendrites and generates an output signal along its axon which makes tree with other axons of other such neurons to form inputs to at the dendrites of subsequent neurons as shown in Figure.2.

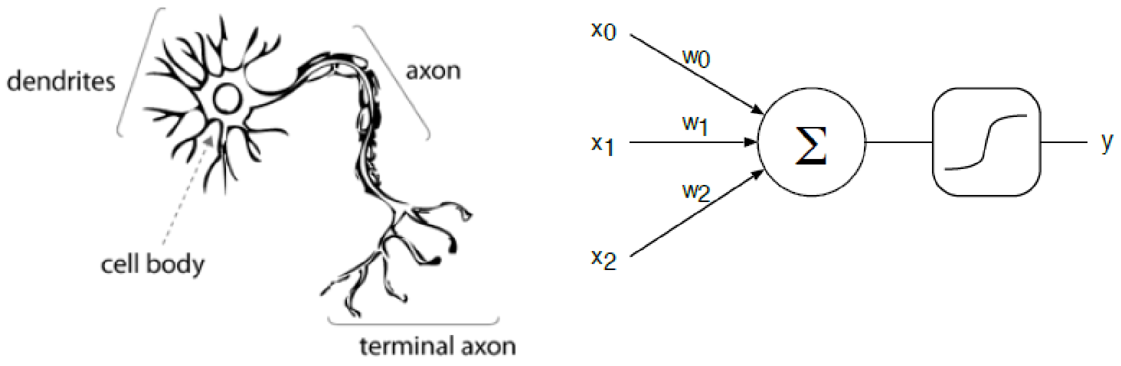


Figure 2.1 Left hand side shows biological neuron and right-hand side shows artificial neuron Image taken from source [2]

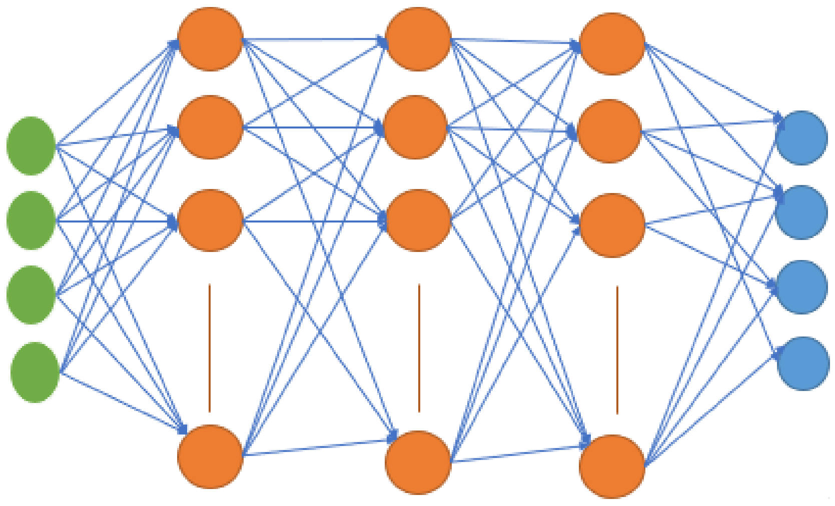


Figure 2.2 Artificial Neural Network

Synapses usually affect the transfer of knowledge from one to another neuron. This is done by affecting a signal along its axon. This scheme further makes branches in outward direction and connects to dendrites of other neurons. Billions of simple neurons connect with each other to form a complex system which enables humans smell, hear, feel, communicate, remember and even fantasize [2], [3].

Artificial Neural Network:

Figure 2.2 shows the building block for an ANN. Artifical neuron receive input signal X_i from other neurons. X_i inputs are multiplied by W_i weights. This mimics the process of synaptic interaction of the dendrites. Input signals Xi are multiplied by corresponding weights values Wi and then summed together with a baised W_b. Resultant value is than fed to non-linear function (activation) which generates output signal as shown in equation 2.1

$$y = f(\sum[Wi * Xi] + Wb) \qquad (2.1)$$

The weights W_i are tuning parameters which defines the behavior of neurons depending on certain type of output [2][4].

A neural network is made by interconnecting millions of such simple artificial neurons which are connected in a subsequent graph fashion which forms a feed-forward Neural Network. Figure 2.2 shows an example of feed-forward neural network. It has one input layer, 3 hidden layers and one output layer.

Network Training:

Neural Network parameters are not engineered using scientific knowledge. These are usually learnt by training these models with inputs and labels. There are two types of approaches to train a Neural Network. First one is supervised learning and second one is unsupervised learning. Supervised learning is most effective of the two types of learning. It consists of inputs with their predefined labels. Optimization of network using supervised learning is carried out by passing it through all training examples at once. This process is called an epoch. Complete training of Neural Network can take anywhere from one to a few hundred epochs depending upon type of data and capacity of network. The training starts randomly initialized weights. Input images are subjected to this network and resultant output is compared against its labels using a loss function. Loss function gives the measure of difference between generated output and corresponding labels. Learning process is geared towards minimizing this loss function on the training inputs and labels by changing the weights.

Stochastic Gradient Descent (SGD) is very efficient optimization technique which is used for optimizing weights in neural networks. SGD algorithm runs on top of neural network training and generates a gradient which relates the influence of each weight on the error. Gradient vectors are calculated by backpropagating output error through the network. The optimization process takes input images, generates loss on these inputs, calculates the gradients, and modify all parameters by a small margin in the direction which is opposite in nature. The strength of these updates is provided through learning rate [2][5][6].

Performance Validation:

Adjusting the weight parameters in each iteration, network converges towards a solution which has minimum loss. Desired outputs are resulted on the training dataset. After each epoch, model's performance is verified with validation dataset. This dataset is not part of training dataset. Actual real-world training dataset usually results in good approximation of network on unseen data. However, if training dataset is very small or model has bigger capacity to learn, the CNN can memorize examples and loses its ability to classify on unseen examples. This problem is called overfitting. This problem can be catered through increase in training samples or by bringing alterations to the network structure like the addition of regularization methods [2].

2.1.2 Convolutional Neural Network

CNNs are special case of ANNs. In ANNs neurons are stacked in 2D arrays or vectors whereas, in CNNs neurons are stacked in multiple dimensions. Like Artificial Neural Networks, CNNs also take input from previous layer and feed to subsequently next layer for further transformations. Like ANNs, each neuron in CNNs also applies a weight to each of its input neuron, and adds the weight multiplied input together with a bias. Result of this addition is then subjected to an activation function which applies non-linearity to this neuron value and limits its output to a reasonable range.

Convolutional Neural Networks (CNN) use small windows of kernel weights which are convolved with a number of input feature maps. In this way neurons are shared across multiple input feature maps. Then the kernel windows are moved over the input feature maps in each layer for the convolution. CNNs also have subsampling or pooling operations. In pooling layer some outputs are taken in a 2x2 or 3x3 window and either maximum value is taken or average of all the values is taken. Kernel size of one make a degenerate case of CNNs in Artificial Neural Network [7][8].

Types of Layers in CNN:

CNN model architecture usually comprises of different types of layers like convolution layer, pooling layer and fully connected layer. Convolutional layer extracts feature of feature maps, followed by Nonlinearity layer and Pooling layer. Pooling layer reduce the size of feature maps. A typical CNN Layers consist of following layers.

Convolution Layer;

Convolutional layer performs convolution of kernels on input feature maps to produce output feature maps. Equation 2.2 and Equation 2.3 gives the dimension of output feature maps depending upon the values of size of kernel window (K), Stride of kernel window (S) and Padding around input feature maps(P). Stride reduces the dimension of the image as follows R_{out} = R_{in}/S. Padding is used for filters whose size is greater than 1x1 as it reduces the size of image. To retain the size of image we use the padding of P = K/2. Taking into account all these parameters, Equation 2.2 and 2.3 give the size of the output feature map

$$\mathrm{Rout} = 1 + \frac{\mathrm{Rin} + 2 * \mathrm{P} - \mathrm{K}}{\mathrm{S}} \qquad (2.2)$$

$$\mathrm{Cout} = 1 + \frac{\mathrm{Cin} + 2 * \mathrm{P} - \mathrm{K}}{\mathrm{S}} \qquad (2.3)$$

Non-Linearity Layer:

NL layer applies nonlinear activation function to each neuron of the input feature map which results from trainable weight layers. Various types of activation functions that are used in Convolutional Neural Network. For example, early on people used sigmoid Equation 2.4 and tanh Equation 2.5. But now they are not used. Because they create a diminishing gradient problem in back propagation. Rectified nonlinear Unit (ReLU) is at the moment most extensively used non-linearity function. It preserves the gradient very well in backpropagation. Equation 2.6 shows ReLU layer operation. ReLU layer cuts off the negative side of the signal. It only outputs positive signal forward. ReLU layer gives the best possible approximations in CNN, that’s why it is used most frequently.

$$\mathrm{f(x)} = \frac{1}{1 + \exp(-\mathrm{x})} \qquad (2.4)$$

$$\mathrm{f(x)} = \tanh\,(\mathrm{x}) \qquad (2.5)$$

$$f(x) = \max(0, x) \qquad (2.6)$$

Figure 2.4 gives the comparison of results of sigmoid, tanh and ReLU activation functions.

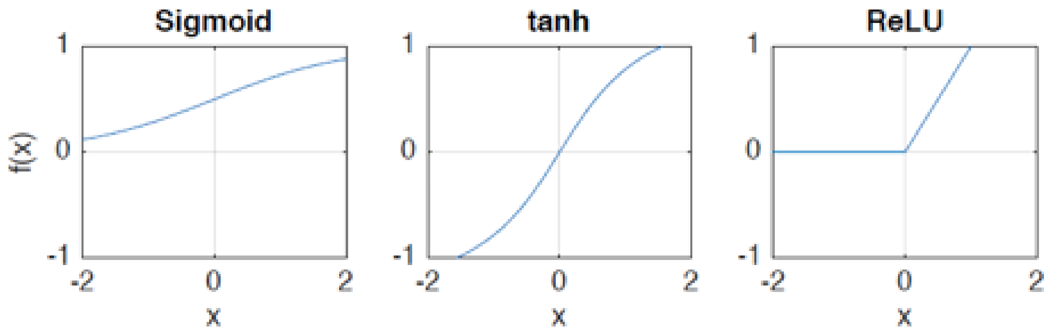


Figure 2.3 comparison of results for different types of activation functions

Pooling Layer:
Pooling or subsampling layer is layer that is used to down sample the input images to output images. Pooling layer reduces the size of image. Pooling layer preserves the most important features in the images. Pooling layer provides scale and distortion invariance to input feature maps. Two types of Pooling function are used in Convolutional Neural Networks, Max-Pooling and Average-Pooling. In Max-Pooling, maximum value is chosen to be output in a certain spatial location. And in Average-Pooling average is taken of all the values of that particular spatial location. A window of size 2x2 or 3x3 is applied at a particular spatial location. Stride is usually the same or lower than size of the patch. Equation 2.2 and 2.3 Apply to get the dimension of Pooled output feature maps.

Fully Connected Layer:
Fully connected layers are Artificial Neural Network Layers. Equation 2.1 gives the details of FC Layers. These layers are usually used at the end of CNN to find the class score in an image classification problem. Most of the weights in convolutional layers are usually from fully connected Layers. Number of weights in a Fully connected Layers is MxN where N is number of Neurons in output layer and M is number of neurons in input layer.

Local Response Normalization (LRN) Layers:
This layer brings competition among different adjacent neurons of an output channels through normalization of their responses. This is done w.r.t a special neighborhood of N channels. AlexNet architecture had first used Local Response Normalization layers[10].

Batch Normalization (BN) Layer:
This layer is applied after completion of each training batch. BN layer normalizes the output of a layer to unit-variance and zero-mean distribution. This uniform distribution to next upcoming layer produces high learning rates and accelerate the training. BN Layers usually result in improved accuracy of the network.

Dropout Layers:
Dropout Layer is best technique to overcome the problem of overfitting in Neural Networks. Dropout layer randomly drops a certain amount of its layer connections during training phase. This technique makes sure that network will not learn precisely. Dropout layer produces abstraction in general behavior of network. it also brings redundancy which is built into the learnt weights.

SoftMax Layers:
SoftMax layer is the most widely used classifier layer used in CNNs. A classifier layer usually follows a fully-connected layer in a CNN. SoftMax transforms the raw scores Z_i of certain class into probabilities P_i as shown in equation 2.7

$$\mathrm{Pi} = \frac{\exp(\mathrm{Zi})}{\sum_{K} \exp(\mathrm{Zk})} \qquad (2.7)$$

This equation results in P which is equal to 1.

2.2 Neural Network Training Frameworks

Various different platforms have been specifically built implementing Convolutional Neural Networks. Some of the examples include MATLAB's Neural Network Toolbox [12], TensorFlow [17], Keras [15], Torch [16] and Caffe [18]. All these frameworks have support for GPUs to accelerate the processing of learning. In this study, Caffe platform is used to build and train CNN models.

2.3 State of art CNN Architectures

Once the most difficult task in Computer Vision is to recognize an object in an image. This is an easy task to accomplish for humans but computers have to go through extensive amount of computations to finalize an object in an image. For example, whether there is cat or dog or car or road in an image. There could be many labels for object in an image or output could yes or no for example is there a person in front of car or not. As an extension of image recognition, scene labeling assigns labels to each and every pixel of an image.

Topologies of Convolutional Neural Networks

Huge number of training images and complexity of the problem presents at ImageNet challenge is an ideal opportunity for researchers and developers to come up with new models and techniques to increase the capabilities of machine vision. Since inception in 2012 convolutional neural networks have emerged as the most successful algorithms in this competition. Top-5 and Top-1 error rates of these CNN based models have reduced very each yearly. Most successful models in last 10 years of ILSRV have been listed below.

AlexNet:

This network was made by Alex Khrizhevsky et al. AlexNet won ImageNet competition in 2012. Emergence of AlexNet algorithm is considered a major achievement in the field of Deep Learning. This network architecture has 5 convolutional layers, it consists of 60.5 million parameters and requires about 1.3 billion MACC operations in single execution of forward pass.

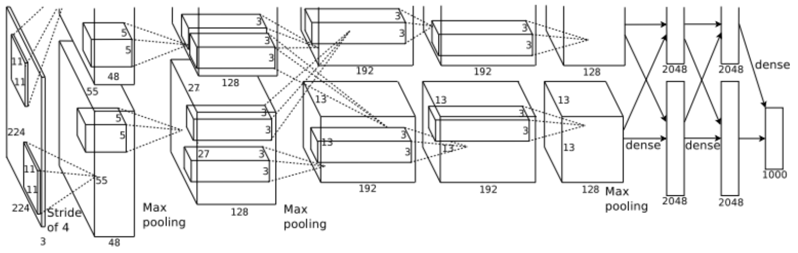


Figure 2.4 AlexNet Network Architecture. Image taken from source [10]

Network-in-Network

This CNN Architecture was made in National University of Singapore in 2013. Network in Network (NiN) architecture is made of multilayer perceptron consisting of small stacks. These perceptions are slid over input feature maps just like convolutional filters. These models have also used global average pooling layer instead of fully connected layers in classifier. Using global average pooling layer in place of Fully Connected layers reduces the number of weight parameters in a network. This network never really participated in ILSRV but it has the accuracy of AlexNet architecture.

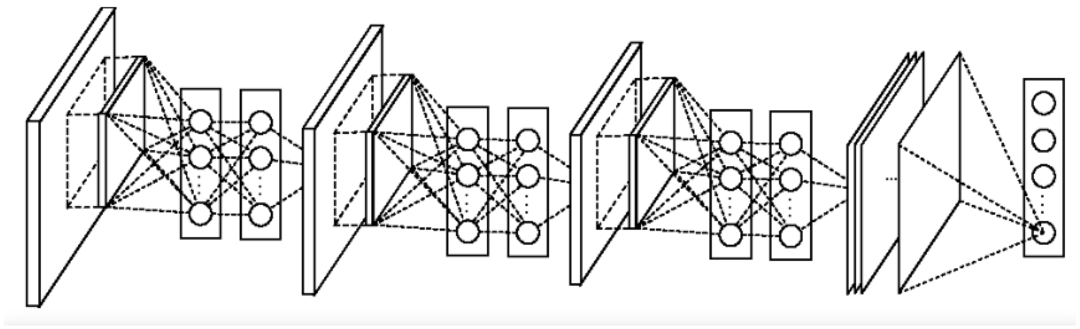


Figure 2.5 NiN Network Architecture. Image taken from source [40]

Visual Geometric Group (VGG)

VGG architecture is named after the group of researchers at University of Oxford. VGG architecture won ImageNet ILSRV in 2014. These networks are very deep usually 18 to 22 layers. Most popular VGG network has16 layers with trainable parameters. This network is usually very regular in structure with convolutions of 3x3 and pooling of 2x2 throughout the network. This network has achieved a top-5 error of about 7.3% which is significantly better than AlexNet. This network contains 140 million weights and it requires 16 billion MACCs. Figure 2.5 shows the architecture details of VGG.

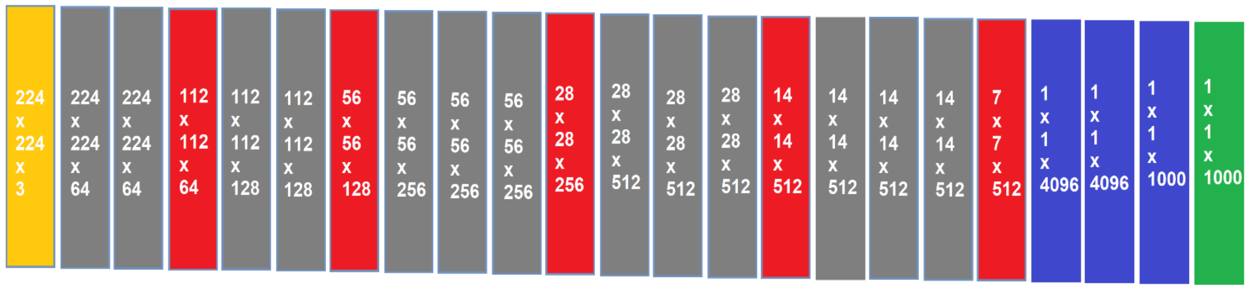


Figure 2.6 VGG-16 Convolutional Neural Network. Input Layer (Yellow), Convolutional Layer (Gray), Pooling Layer (Red), Fully Connected Layer (Blue), SoftMax Layer (Green)

GoogLeNet

This network was made by researchers at google. This is considered as a major work in the domain of Machine Learning. It was published just a few days after VGG-16 Network emerged. This network is even deeper than VGG, it consists of 22 layers. GoogLeNet has achieved 6.67% error in top-5 category.

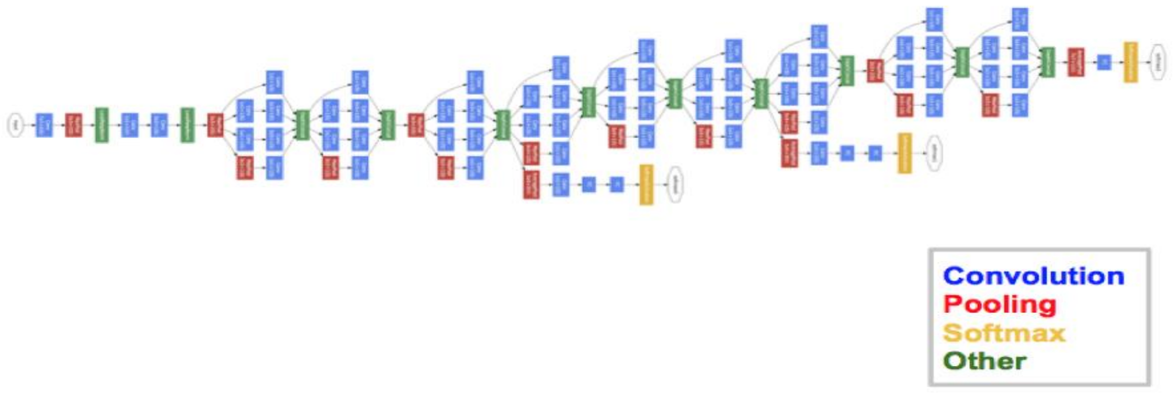


Figure 2.7 GoogLeNet Network Architecture. Image taken from source [58]

ResNet

This network architecture was made by Microsoft Research group. It won the ILSVRC competition in 2015. Previously, architectures which had more than 25 layers with trainable weights were considered hard to train. To solve this problem designers introduced detours around the batch of two convolutional layers. These two detours were than summed actual and the one which were filtered for representation together at a certain point. ResNet achieved an error of 6.7% in top-5 category [53].

SqueeezeNet:
This architecture was developed at UC Berkeley, research paper [44] published in February 2016. SqueezeNet architecture is different from the other CNN architectures in design goal. As it was made to reduce number of trainable parameters in network not to increase accuracy. Authors in this work managed to develop a network which had an accuracy of AlexNet, but with 50× less parameter. Fire modules introduced in this model resulted in parameter reduction. Fire module is a reduce-expand micro architecture which is similar to the Inception modules. Fully-Connected, LRN and Batch Normalization layers were omitted from architecture.

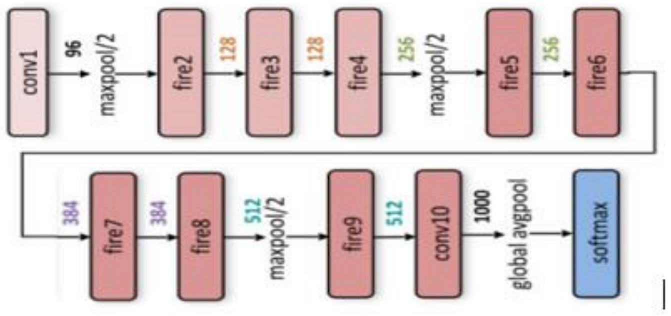


Figure 2.9 SqueezeNet Network Architecture [44]

CHAPTER 3: FIELD PROGRAMABLE GATE ARRAYS

In this chapter we discuss about Field Programmable Gate Arrays. First, we introduce FPGA and how they compare with other computing platforms, highlighting key characteristics, strengths and weaknesses of this platform. This chapter also gives introduction on basics of HLS. It is a new design framework which enables to configure FPGAs in higher abstraction level languages such as SystemC/C/C++. And at the end we discuss about all previous implementations are CNN on FPGA.

3.1 Introduction to Field-Programmable Gate Arrays

FPGAs are semiconductor devices which consist of two-dimensional arrays of reprogrammable logic blocks also called logic slices. Logic slices in FPGA are connected through programmable interconnects. Interconnects are collection of wire bundles which are running horizontally and vertically between logic slices. Each interconnect has a switch box. Modern FPGAs usually consist of millions of such Logic Slices. Modern FPGAs also has a lot of functional units which makes it possible to implement common arithmetic functions fast and efficiently. Configuration bitstream is loaded into device which configures logic slices, arithmetic function units and interconnects. FPGAs can be programmed many times [22][23].

FPGAs Versus CPU Computers: The biggest benefit that an FPGA Platform can provide compared to general-purpose computers such as desktop computers, smartphones and Graphical Processing Units is the availability of programmable hardware which consists of general-purpose logic blocks, look up tables DSP slices etc. Programmable hardware resources available in FPGA allow to us to make special architecture to carry out specific tasks. This results in high energy efficiency, high throughput and speed. This advantage of FPGAs usually comes at the expense of reduced programmability and enhanced complexity during period of development. Designers usually need to take into consideration the available hardware resources on given platform and the efficient implementation of algorithm into FPGA architecture. Some algorithms might not be able to map well at all on FPGA considering rigid block structures found on FPGA [22][24].

FPGAs versus ASICs: ASICs are custom built chips made with semiconductor devices. Compared to FPGAs, they do not exhibit any area or timing delays from configuring logic and interconnects. Hence, they usually result in most energy efficient systems. But ASICs Chips are manufactured through sophisticated fabrication process which can take long development cycles and very high NRE costs. Design and development of ASIC systems demands a lot of testing and verification before manufacturing and details on each step of development. So, they provide a good solution for cost-sensitive and high-volume applications. FPGAs with their reprogrammable are more suitable for making new application in short time [22].

3.2 Introduction to High Level Synthesis

Xilinx Vivado High level synthesis (HLS) tool converts a code written in C/C++ into register transfer level (RTL) implementation which can be synthesized into an FPGA. Historically, FPGAs have been traditionally programmed with CHDL or Verilog Languages. Designs are written mostly at RTL. At RTL level the designer specifies algorithm to carry it out in parallel. So, vector processing is key element to achieving high design throughput. RTL design is a process of describing combinational logics, basic arithmetic operations and registers. These RTL designs are managed by the falling and rising edges of a clock signal. These design specifications are very close to the logic gates and wires like in RTL which are available in the provided FPGA or ASIC technology. Generated hardware from RTL Synthesis can also be controlled. Breaking down an algorithm into combinational logic, arithmetic operations and registers on RTL can a be tedious and error-prone and time-consuming work. A lot of decisions during design process are made before writing any code. Changes made at later stages can be difficult and costly [22].

The promise of high-level synthesis (HLS) is welcome development as HLS has the ability to produce very good register transfer level (RTL) implementations from high-level descriptions like C, C++ or SystemC. So, we can say that HLS automatically transforms manual process of writing huge RTL. It reduces the sources of a lot of design errors and also speeds up a very long and iterative part of the development cycle.

Increased Level of Abstraction with HLS: Instead of coding FPGA RTLs in Verilog and VHDL, High-Level Synthesis (HLS) provides another easy to adopt HLS. It helps us to write RTL codes in a higher abstraction level language such as C, C++ or SystemC. In doing so, minor details of design implementation can be taken away and accomplished by the Vivado HLS compiler, which then changes the software written in C, C++ or SystemC into RTL specification.

High Level Synthesis (HLS):

Xilinx's VHLS is major development. Vivado HLS enables Engineers to can use structs, floats, loops, arrays, arithmetic operations and functional calls. HLS compiler than automatically converts these into memories, computational cores, counters and handshake protocols. This process also results in associated state machines and schedules. The compilation of program is influenced through compiler directives or compiler PRAGMS, meta instructions that only an HLS compiler can comprehend. By default, operations are scheduled to be performed as early as possible. With these compiler pragmas, the engineer can further influence the designs at any level like memories, loops and pipelines [22][23].

Promises and Difficulties: The increased level of abstraction in High-Level Synthesis enables us to achieve Fast development rounds with optimizations and productivity which is much higher. Though, this comes at the cost of lesser control on the end product. Keeping in mind latest trends in market especially with regards to design complexities which keep on increasing and time-to-market keep on decreasing. This turns out to be a promising approach to design space exploration. Even after being so impressive in increasing productivity of FPGA based designs people working in R&D still prefer Verilog or VHDL. It is extremely difficult job to convert sequential programs into parallel executing entities with same number of optimizations and performance. Even though different firms have invested billions of dollars and many years of research into High Level Synthesis [22], [23], [24]. Performance and efficiency of program written in HLS highly depend upon style of coding and finest of details which have to be done in Verilog and VHDL. There are still some flaws in High Level Synthesis and it can only be discovered at compile time. So, the decision our decision to use High Level Synthesis is associated with non-negligible risk [25].

3.3 Embedded Convolutional Neural Networks

This section provides a discuss on implementation of CNNs on various platforms like CPUs, GPUs and ASICs leading up to the implementation on FPGA. It is common practice among community to train model on high performance GPUs based systems and then deploy trained model on different embedded systems. The following sections describes important implementation of CNN algorithm on different embedded platforms.

3.3.1 Potential Hardware Platforms

Embedded designs usually comprise of very specific design goals such as power, energy, speed reliability and battery requirements. These design goals can make deploying deep learning models on these embedded systems a very different field of research. As there are a lot of different types of embedded systems which can be deployed to carry out the designs. So, of them are listed below with details of some sort.

Central Processing Units (CPUs): CPUs are general purpose computers which can be found in a lot of devices these days. Most of them consist of personal computers and smartphones. These CPU computers are hugely flexible in nature and can carry out wide variety of workloads. There are a lot of different processors available today for embedded systems design and implementation. All of them have different tradeoffs regarding speed and power requirements. But downside to these CPUs is that they are sequentially computing systems and don’t allow parallelism.

Digital Signal Processors (DSPs): DSPs are highly specialized micro-processors. These processors are optimized for computing floating-point signals very fast and efficiently. DSPs have VLIW instructions set. It helps to increase parallelism in code. Modern DSPs processors can contain multiple cores which run in GHz with peak DSP performance up to 160 GFLOP/s at less than 15W. but these processors are not suitable for implementation of CNNs.

Graphics Processing Units (GPUs)

GPUs are multiple-core computers. They were originally made to perform highly parallel algorithms in Graphics based applications. Recently, GPUs have been used for general-purpose computing tasks as well. A high-end GPU can contain more thousands of floating-point processing cores which can run at up to 1 GHz of frequency, with a bandwidth of 330 GB/s. They can have a peak DSP performance of up to 6600 GFLOP/s. They can consume up to 250W of power. GPUs are most suitable because of nature of workload presented by CNNs and are fully supported by most deep learning frameworks. They also become the major platform for research in the area of CNNs.

Field-Programmable Gate Arrays (FGPAs):
Already introduced in section 2.2. FPGAs can contain millions of logics cells, and thousands of DSP units. Run at frequency of 300MHz and can generate a peak floating-point performance of 1000 GFLOP/s at a few tens of watts [26], [27], [28]. But FPGA work best on algorithms which have regular computation patterns. And thus, parallelism can be exploited using the programmable logic blocks. Algorithms with huge data-dependencies are simply not suitable for implementation on FPGA.

Application-Specific Integrated Circuits (ASICs)
In terms of energy efficiency and performance gains ASICs provide usually the best solutions. But compared to FPGA they are even more less suitable for irregular computation patterns. That is why ASICs chips are typically made only to accelerate a certain aspect of CNNs.

3.3.2 Existing CNN Implementations

Accelerating Datacenter Workloads using FPGAs:
Microsoft and Baidu have built FPGA-based accelerators for their search engines workload in datacenters. Microsoft’s Catapult platform [42] has already doubled the speed of Bing ranking algorithm. Recently, it was deployed to implement a record breaking AlexNet accelerator which achieved a throughput of x2 times compared to modern GPU based system at 1/10 of the power consumption. Baidu made similar plan to use FPGA and they deployed Altera based system.

ASIC Implementations: DaDianNao (2014) is an accelerating system consisting of 64 ASIC chips. These ASIC chips have large on-chip memories to reduce off-chip memory traffic resulting in optimize energy efficiency. Based on their simulation results, paper claims off up to 450 times better performance and 150 times lesser energy budget with respect to a GPU implementation [46].

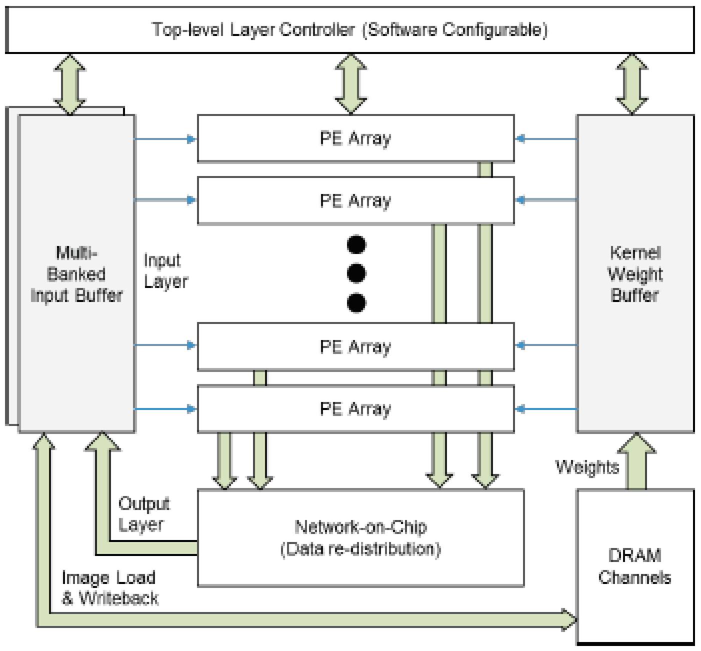


Figure 3.1 Top-Level Overview of the FPGA-based CNN Accelerator developed by Microsoft [42]

FPGA based CNN accelerator have been a hot topic in last decade. Different people have used different approaches to implement CNN on FPGA. In all of these works training of convolutional neural network was performed offline and trained model was deployed on FPGA. Convolutional layer is computationally expensive layer in CNN, while Pooling layer, Fully Connected layers and SoftMax layers are less expensive. So, in most of these works' convolution was performed on FPGA while other tasks are performed with Host PC.

Authors in [33] implemented a facial detection system using low-end DSP-oriented Field Programmable Gate Array (FPGA). Limited capacity of FPGA didn't allow them complete design space exploration. Network was implemented using systolic architecture with a 7x7 convolution window doing the filtering job and Host PC doing other tasks in CNN. This work achieved a kernel level parallelism in execution of Convolutions, and achieved a peak performance of 3.4 GMACC/Second for Convolutions implemented in FPGA overall achieved a frame rate of 6 for scale and distortion invariance network.

Work [34] is the extension of work [33]. This work explored further parallelism in convolution. Implemented network to achieve inter output parallelism, where one input feature maps make multiple output feature maps. They used 4 convolution windows of size 5x5. Prototype of the CNN accelerator was implemented on Xilinx Virtex5 LX330T FPGA with four DDR2(total of 1GB) external memory banks. The Accelerator prototype could process at the rate of 3.4 GMACCs for CNN forward propagation, a speed that is 31x faster than base software implementation on a 2.2 GHz AMD Opteron processor.

Authors in [35] made four categories of workloads in CNN, and found that most computationally expensive part of algorithm can be formulated as matrix-vector or vector-vector multiplication. Operations of CNN were divided into various steps, matrix-vector or vector-vector operations usually result in large amounts of intermediate data, in second step this large intermediate data was further reduced by a secondary layer of operations such as finding max/min, array ranking and aggregation. Their accelerator, which they called MAPLE, consists of hundreds of such processing units laid out in a 2D grid, with two key features. First, Accelerator used on-chip memory blocks to store large intermediate data perform to perform secondary reduction operations. So, this scheme decreased in off-chip memory traffic and consequently increased performance. Secondly, Accelerator organized off-chip memory into different banks and program its processing units read off-chip data independently. These two features combined together allowed accelerator to improve its performance with large data size. This work further explored its design space and illustrated how application kernels can be mapped to the hardware automatically. Implemented a 512-PE FPGA prototype running at 125 MHz of Accelerator which was 1.5-10x faster than a 2.5 GHz quad-core Xeon processor.
Work [36] is the extension of work [33][34][35]. This work proposed a CNN hardware architecture which could dynamically configure its hardware in runtime to match different types of parallelisms that exist in CNN workload. In this work a CNN compiler was used which could automatically translate a high-level abstraction of CNN specification into a parallel low-level microprogram. This architecture was then mapped into FPGA. It was scheduled. The Host processor executed the operations mapped on hardware. Compared the performance of proposed architecture to a 2.3 GHz quad-core, dual socket Intel Xeon C870 CPU @1.35 GHz and also with a 200 MHz FPGA implementation. Accelerator running at120 MHz dynamically configurable architecture was recorded to be 4x to 8x faster. This was the reportedly first CNN hardware architecture which could achieved real-time video processing on a variety of object detection and pattern recognition tasks.
Authors in [37] discussed the external memory patterns and proposed an accelerator design which could give optimum performance for given CNN workload. This paper explained issues which were mostly related to limited amount of DDR memory bandwidth provided in FPGA Platforms. Further this work introduced highly flexible memory hierarchy which could minimize the effects of the memory bottleneck in complex memory access patterns of CNN workloads. Efficiency of the on-chip memories was maximized by scheduler that used tiling to optimize for data locality in convolutional layers. Accelerator design also ensured to keep on-chip memory size small, which in turn could reduce area and energy usage. Proposed architecture was evaluated by a High-Level Synthesis implementation on a Virtex 6 FPGA board. This accelerator achieved a performance that was 11x faster than previous implementation.
These implementations build FPGA-based Accelerator based on systolic architectures and run this accelerator with host PC through software. Systolic architectures are good solution for algorithms which have performance as bottleneck. CNNs have complex memory patterns which makes memory access a bottleneck as well. So, for implementing CNN systolic architectures don’t guarantee high throughput for complete operation of CNN. Any

performance gain achieved on FPGA is reduced on host PC when reading or writing feature maps.
Work [38] and [39] is complete hardware implementation of DCNN. They have mapped complete convolution on FPGA. Beside previous work only focusses on filtering and convolution this work also takes into account memory access patterns and optimize them to get good result. Authors in [38] discussed a critical problem that the computation throughput of CNNs did not match the external memory bandwidth provided by current FPGA platform. So, previous approaches to design space exploration could not achieve high enough performance because of under-utilization of either logic resource or eternal memory bandwidth. To overcome this issue, an analytical design scheme was discussed using the roofline model. For any solution of a CNN design space exploration, quantitatively analyze computing throughput and required external memory bandwidth. This is done using different kind of optimization techniques, such as loop tiling and loop transformation based on dependency analysis. Then, with the help of roofline model identify the solution which has best performance and lowest FPGA resource requirement. Authors mapped a CNN accelerator which was only for convolutional layers. Design space exploration was carried out on a VC707 FPGA board. They compared the result to previous approaches. This mapping scheme could achieve a peak performance of 61.62 GFLOPS at 100MHz working frequency, which latterly outperformed all previous approaches significantly.

[39] Proposed an FPGA-based accelerator architecture which could leverage all sources of parallelism in CNNs. They developed analytical model to check the feasibility and performance estimation. This analytical model took into account various design and platform parameters. They also proposed a design space exploration scheme for obtaining the implementation which resulted in highest performance on any platform. Their simulation results with state-of-the-art CNN demonstrated that proposed accelerator could run 1.9x faster than state of the art CNN accelerator on the same FPGA device [38].

CHAPTER 4: CNN IMPLEMENTATION

This chapter presents the detailed discussion on already existing CNN Topologies, discussing meanwhile training platforms as well as optimization which we build into our CNN architecture. First, the CNN architectures were completely studied from prior work. Secondly, details discussion is presented on choosing a particular network depending upon the platform that we have at our disposal.

4.1 VGG-16 Network

Different network architectures were discussed in chapter.2. State of art network architectures include SqueezeNet, GoogLeNet, Visual Geometric Group, Network-in-Network, ResNet, Inceptions and AlexNet. Figure 4.1 present an analysis of all these networks. On left is top-1 accuracy and on right is top-1 accuracy of all these networks with respect to number of operations (GOPs) in each of these networks. Graph on Right shows the compute intensive nature of CNN algorithms.

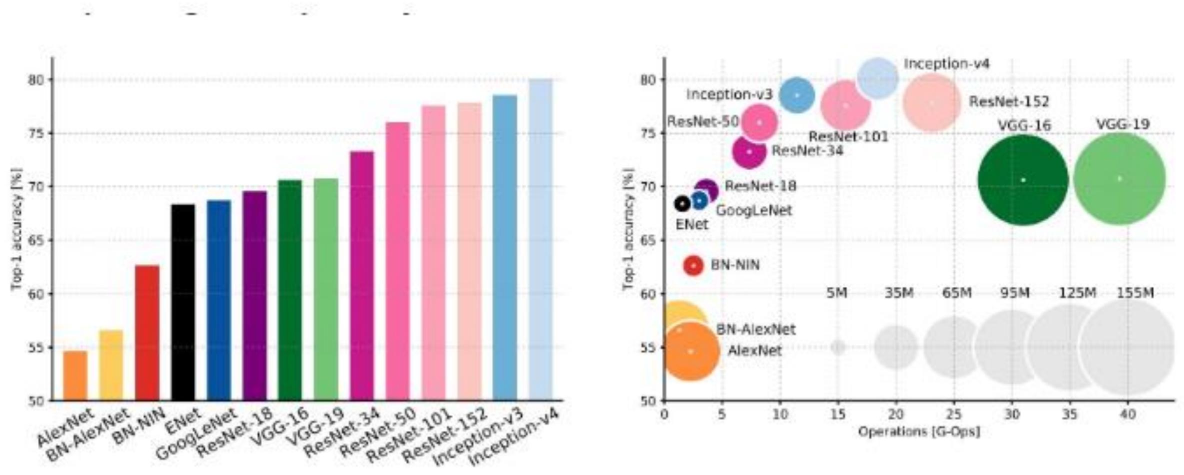


Figure 4.1 Analysis of Convolutional Neural Network Architectures for Practical Applications. Image taken from source [59]

Table 4.1 shows the properties of these networks. From Table 4.1 we can see that Convolutional Networks require huge amount of memory and perform millions of computations. There have been various implementations of these networks most notably on general purpose computer. Compared to all the networks, VGG- 16 style networks are most admired and used architectures in community. These networks have uniform structure. Convolutional kernel size (3x3) and Pooling Window(2x2) remains same throughout the network. All convolutional layers have same stride (1) and padding ('same').

Table 4.1 Comparison of characteristics of Different Network Architectures

Network Architectures	**#conv layers**	**No of MACC (million)**	**No of Param (million)**	**No of Neurons (million)**	**Top-5 Error Rates**
AlexNet	5	1140	62.4	2.5	19.7%
Network-in-Network	12	1100	7.6	4	19.0%
VGG-16	16	15470	133	29	8.1%
GoogLeNet	22	1600	7.0	10	9.2%
ResNet-50	50	3870	25.6	47	7.0%
Inception-v3	48	5710	24	32.6	5.6%
Inception-Resnet-v2	96	9210	31.6	75	4.9%
SqueezeNet	18	860	1.2	12.7	19.7%

All pooling layers have stride of (2). This study considers the implementation of VGG-16 style networks on FPGA. State-of-the-art CNNs require huge amounts of memory for storing their weights and neurons. VGG-16 would require approximately 700MB of DDR3 with 32-bit floating points precision. Proposed accelerator was intended to be developed on Xilinx Zynq Zedboard which has only 512 MB on chip DDR3 memory. Fitting a complete VGG-16 network is impossible task. So, a smaller variation of VGG- 16 style network is used. Instead of training VGG-16 for 1000 classes a smaller network for 5 classes is trained and deployed. Kaggle’s Flower recognition example is used [55].

Table 4.2 Network Configurations of VGG-16 Style Network

Layers	**N**	**R**	**C**	**M**	**K**
Input	-	150	150	3	0
Conv1	3	150	150	32	3
Pool1	32	75	75	32	2
Conv2	32	75	75	64	3
Pool2	64	37	37	64	2
Conv3	64	37	37	96	3
Pool3	96	18	18	96	2
Conv4	96	18	18	96	3
Pool4	96	9	9	96	2
FC1	7776	-	-	516	-
FC2	516	-	-	5	-

This example has dataset consisting of 4242 labeled images of five different categories of flowers; Sunflower, Tulip, Daisy, Rose, Dandelion. Each class has approximately 800 images. Images are not high resolution (320x240). Training was carried out on Network shown in Table 4.2 on this dataset. This network architecture has four Convolutional Layers. Each of these layers are subsequently followed by Max Pooling Layers. Each Convolutional Layer Has Rectified Linear Unit (ReLU) activation. There are two Fully Connected Layers. First Fully Connected Layer is followed by ReLU activation and second one is followed by SoftMax layer.

4.2 Training Platforms

Currently There are various different platforms being used to train and deploy convolutional neural networks. Almost all of these platforms are helpful in sense that one doesn't have to go through a lot of coding. Instead focus should primarily be on quality of data the one can have to train our model. At the back end each of these platforms use same kind of mathematics. All of these frameworks help in quick modeling and training of Convolutional Neural Networks. Some of the key feature that any Deep Learning frameworks should have are listed below.

- Performance Optimization
- Easy to understand
- Easy to write code
- Community Support
- Parallelize the processes to reduce execution time

Any deep learning library or platform that has above qualities is regarded as good platform. Since Deep CNNs are hot topic recently, many companies have come forward to make their own platforms. Some of the most famous deep learning frameworks are explained below.

TensorFlow:
This platform is considered as the most popular platform. It was created by Google. It is written is C++ and Python. Big companies like Uber, AirBnB, DropBox, DeepMind all have persuaded to leverage this platform. TensorFlow has highest rate of entry for beginner. This is relative low abstraction level language with a lot of details.
Some of the most popular case of TensorFlow usage are

- Language detection and text summarization
- object detection, Image recognition and facial recognition
- Sound pattern recognition
- Video stream analysis

Keras:
Keras is high end API that runs on top of TensorFlow, Theano or CNTK. It was developed by a Google engineers. Keras is modular framework. It is relatively easy to code and deploy. It has a plenty of layers to work with. It is also very easy to adopt for beginners. It has support for variety of embedded platforms. Some of the key advantages of this framework are

- Fast and easy Prototyping;
- Lightweight for building deep learning architectures
- It has lot of layers
- Its modules are fully configurable
- Easy to use for those entering the field of Deep learning because of it simple and meaningful interfaces
- Since it is based on TensorFlow back end, it has very good support for GPU based systems
- Helps to train DL models multiple GPUs and Google Cloud as well.
- This framework support modeling and training for NVidia GPUs, Google's TPUs, and Open-CL based GPUs such as AMD.

PyTorch:
PyTorch is considered the most famous of all the deep learning frameworks. It is part of the very useful Torch Deep Learning framework. PyTorch runs on Python. PyTorch is a Python package which helps Tensor computations. It also uses dynamic computation graphs. Some of the key advantages of this framework are

- Simple and transparent model networks
- Like traditional programming languages it also has default mode called as define by run
- It has commonly used debugging tools such as PyCharm debugger, ipdb, pdb
- It supports data level parallelism;
- It has a plenty of pretrained models
- It has readymade modular entities which are easy to deploy in our models
- PyTorch like Keras also supports training on multiple GPUs or Clouds.

Caffe:
Caffe was developed by Yangqing Jia during his Ph.D. tenure at the University of California, Berkeley. This framework in written in C++ and it has a Python interface PyCaffe as well. Caffe is mainly build to model and deploy CNNs and various other feed-forward networks. Caffe is considered the most useful framework for training deep learning models. like Keras and PyTorch, we don't have to write codes to model Convolutional Neural Networks. It also has multiple built-in image processing modules. The framework is admired for following reasons:

- Caffe provide a lot of pre-trained models for building DL applications
- It works well with other frameworks as well.
- It has server optimized inference
- It's fast, scalable, and lightweight

Deeplearning4j:

Deeplearning4j is written for Java based frameworks. It is commercial grade and open source framework. It provides very good support for different types of neural networks like CNNs, RNNs, RNTNs, or LTSMs. It has very good potential for developing image processing pipelines and machine learning based natural language processing. Some of the key benefits of this framework are

- Flexible and robust;
- Good performance for handling big data
- It has very good documentation
- It has both community version and enterprise version.

4.3 GPU-based Training System

We set up our training environment on Core i9 (9th Gen) PC with Titan GeForce GTX 1660 GPU. GeForce GTX 1660 is latest generation of GPUs made NVIDIA. They have immense computing power.

We set up Caffe Framework on this system to train our network. Caffe is written is C++ and it has Strong Support for GPU. Caffe comes with option to train on both CPU and GPU. With Caffe one doesn't have to write a lot of codes. Some of the key advantages of using Caffe framework for our network training are as follows.

- Caffe is the fastest frame work for implementing Convolutional Neural Network.
- It has Expressive architecture that enables application innovation and development.
- Models and optimizations can define by writing configuration files
- Switching between CPU and GPU is really easy. It can be done by simply raising a flag to train on a GPU.
- Extensible code peruses active research and development.
- Caffe is being used by 1000 of researchers.
- Speed is what Caffe is all about. Speed make Caffe good platform for research-based experiments and industry design.

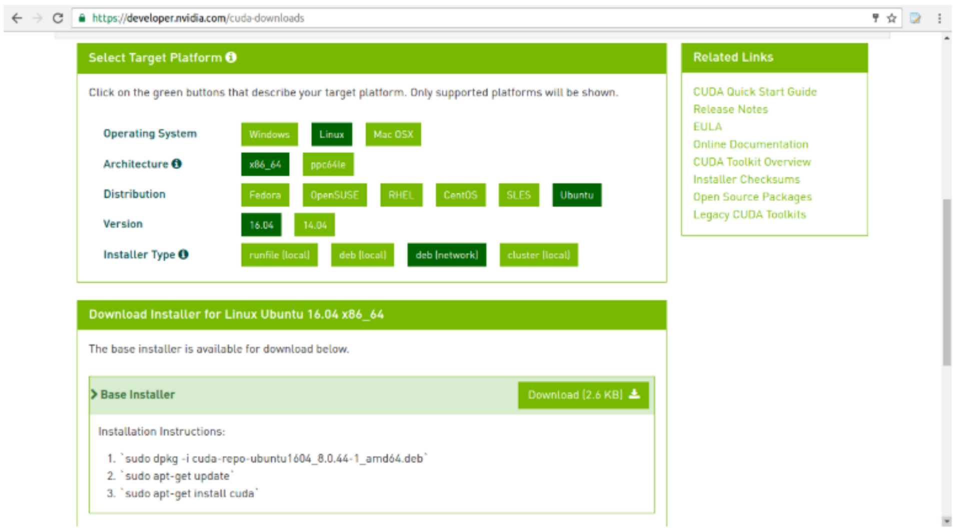
- Caffe has great community support. It has already started to power projects in academia and research, startup projects, and even large-scale industrial projects in vision, speech, and multimedia.

4.4 Setting up GPU-based Training System

There are two modes in which Caffe can be installed on Ubuntu. GPU mode and CPU mode. Since we have GPU based system, Caffe was compiled for GPU option. To install Caffe first install OpenCV and all other dependencies that Ubuntu based system requires to run Caffe.

Some of the key steps in setting up environment are listed below. Ubuntu terminal commands to carry out all the steps to install Caffe on Ubuntu are listed in APPENDIX.A

1. Update and Upgrade Ubuntu through terminal command line. (Command.1-2)
2. Installing required dependencies for OpenCV (Command. 3)
3. Install BLAS Library for computation optimization (Command.4)
4. Install PIP which is useful for handling Python Packages (Command.6-7)
5. Install Python Development Package tools (Command.8)
6. Both OpenCV and Caffe require NumPy to handle arrays and big data, so install latest version of NumPy (Command.8)
7. In Next Step, download and install OpenCV and its contributions. Compile OpenCV and its contributions and install it (Command.9-16)
8. Check whether OpenCV was installed correctly (Command.17-20)
9. With OpenCV set up, one can proceed to install Caffe with GPU support. To install Caffe with GPU support, first correct version of CUDA Development Kit needs to download and install. Figure 4.2 provides the detail of installing CUDA Development Kit.
10. Next, cuDNN downloaded and installed, cuDNN is a GPU-accelerated library of primitives for deep neural networks provided by NVIDIA. cuDNN enhances the computation speed of GPU based computation. To Install it, one needs to go to cuDNN home page, register and download and that install it. Figure 4.3 gives the detail of installation of cuDNN.
11. After successfully installing CUDA and cuDNN, Caffe is installed. To install for GPU, one needs to install all the necessary packages and libraries (Command. 21)
12. Now get the latest version of Caffe from GitHub repository (Command.22)


https://developer.nvidia.com/cuda-downloads

Select Target Platform

Click on the green buttons that describe your target platform. Only supported platforms will be shown.

Operating System	Windows [✓ Linux] Mac OSX
Architecture	[✓ x86_64] ppc64le
Distribution	Fedora OpenSUSE RHEL CentOS SLES [✓ Ubuntu]
Version	[✓ 16.04] 14.04
Installer Type	runfile (local) deb (local) [✓ deb (network)] cluster (local)

Related Links

CUDA Quick Start Guide
Release Notes
EULA
Online Documentation
CUDA Toolkit Overview
Installer Checksums
Open Source Packages
Legacy CUDA Toolkits

Download Installer for Linux Ubuntu 16.04 x86_64

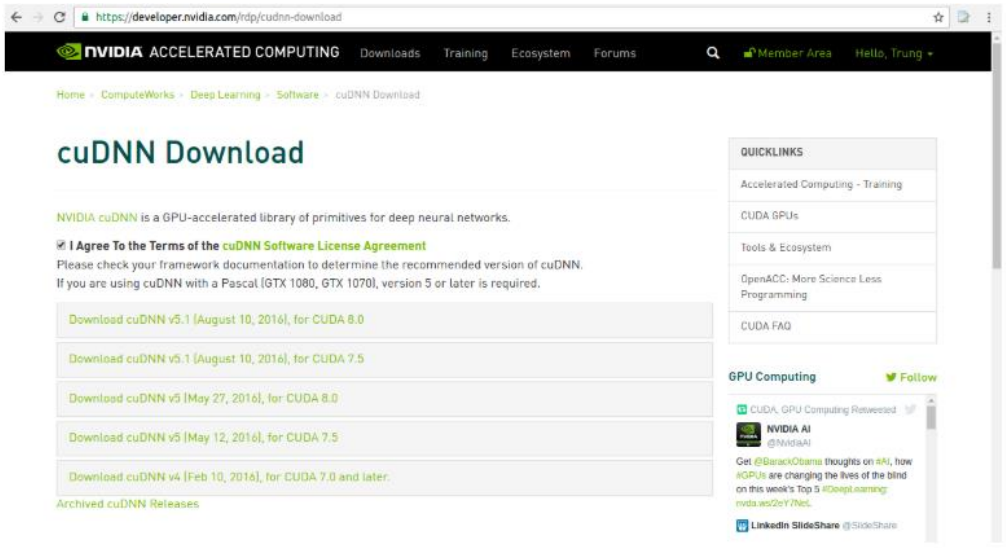
The base installer is available for download below.

Base Installer Download (2.6 KB)

Installation Instructions:

1. `sudo dpkg -i cuda-repo-ubuntu1604_8.0.44-1_amd64.deb`
2. `sudo apt-get update`
3. `sudo apt-get install cuda`

Figure 4.2 Installation of CUDA Development Kit


https://developer.nvidia.com/rdp/cudnn-download

NVIDIA ACCELERATED COMPUTING Downloads Training Ecosystem Forums Member Area Hello, Trung

Home › ComputeWorks › Deep Learning › Software › cuDNN Download

cuDNN Download

NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks.

[✓] I Agree To the Terms of the cuDNN Software License Agreement
Please check your framework documentation to determine the recommended version of cuDNN.
If you are using cuDNN with a Pascal (GTX 1080, GTX 1070), version 5 or later is required.

Download cuDNN v5.1 (August 10, 2016), for CUDA 8.0
Download cuDNN v5.1 (August 10, 2016), for CUDA 7.5
Download cuDNN v5 (May 27, 2016), for CUDA 8.0
Download cuDNN v5 (May 12, 2016), for CUDA 7.5
Download cuDNN v4 (Feb 10, 2016), for CUDA 7.0 and later.
Archived cuDNN Releases

QUICKLINKS

Accelerated Computing - Training
CUDA GPUs
Tools & Ecosystem
OpenACC: More Science Less Programming
CUDA FAQ

GPU Computing Follow

CUDA, GPU Computing Retweeted
NVIDIA AI @NvidiaAI
Get @BarackObama thoughts on #AI, how #GPUs are changing the lives of the blind on this week's Top 5 #DeepLearning: nvda.ws/2eY7NeL
LinkedIn SlideShare @SlideShare

Figure 4.3 Installation of cuDNN Library for Caffe

13. Apply required modifications in configuration file makefile.config (Command.23-31)
14. After that proceed through compilation of Caffe (Command.32)
15. After compilation of Caffe is completed. Check whether its working correctly (Command.33-35)

4.5 Dataset

After setting up the environment network training was carried out. First, dataset needs to be downloaded to train a particular network. As case study, this study uses Kaggle’s flower recognition example. This example has dataset consisting of 4242 labeled images of five different categories of flowers; Sunflower, Tulip, Daisy, Rose, Dandelion. Each class has approximately 800 images. Images are not high resolution (320x240). Network shown in Table 4.2 was trained on this dataset. This network architecture has four Convolutional Layers. Each of these layers are subsequently followed by Max Pooling Layers. Each Convolutional Layer Has Rectified Linear Unit (ReLU) activation. There are two Fully Connected Layers. First Fully Connected Layer is followed by ReLU activation and second one is followed by SoftMax layer.

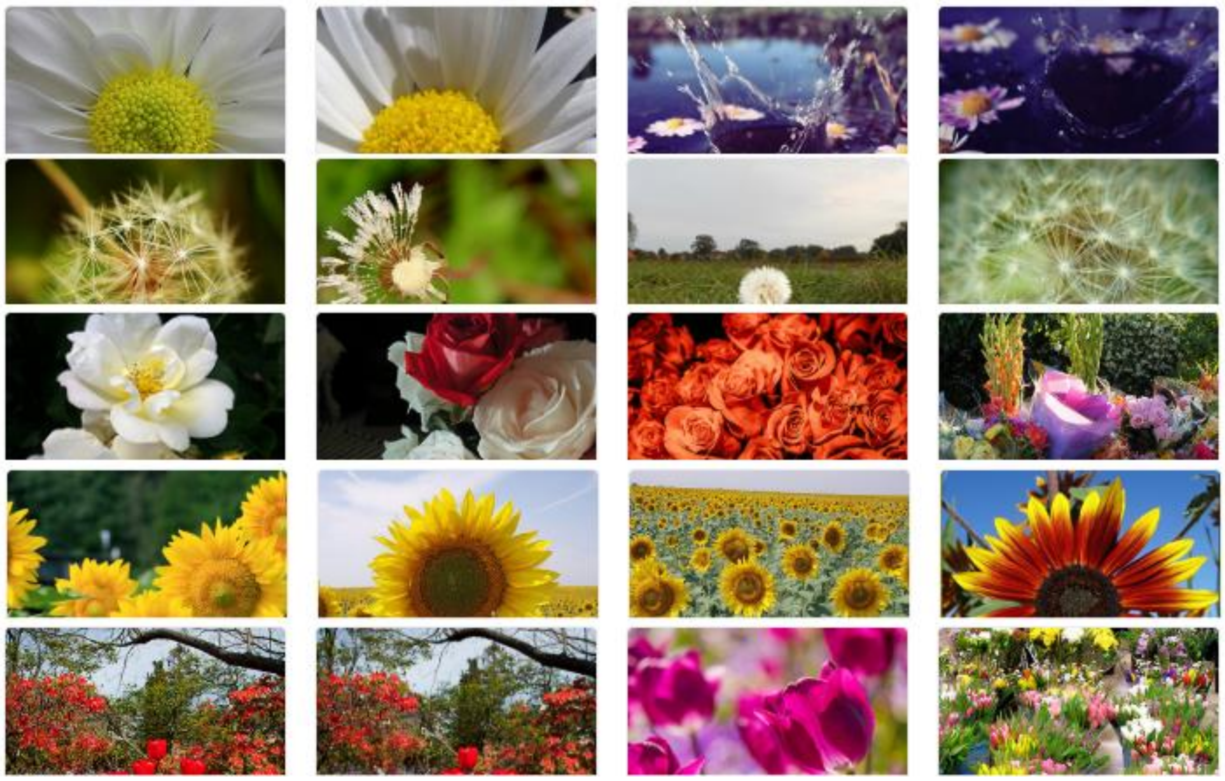


Figure 4.4 Sample images of Kaggle Dataset for Flower Recognition

For Classification of images there are two phases machine learning algorithm:

- **Training phase:** This a phase where a machine learning model learns from given dataset of images. Each image has a label.
- **Prediction phase:** In prediction phase, trained model is utilized to predict labels of images that are not present in training dataset.

The training for an image recognition is done in two main steps:

- **Feature Extraction**: Feature Extraction is done by utilizing the knowledge in that particular domain to extract new features from images which will be used by the learning model.

- **Model Training**: In training phase, a clean dataset is used which is composed of the features of images against labels which trains the machine learning model

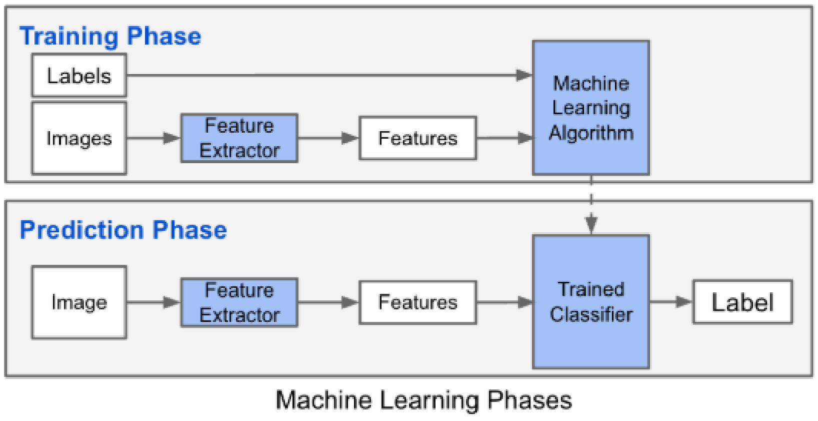


Figure 4.5 Traditional Flow of Machine Vision Algorithm

Figure 4.5 shows a typical machine vision pipeline. The main difference between a machine vision and deep learning pipelines is feature extraction methodology. In machine vision algorithms, features are handcrafted using different mathematical tools. Whereas, in traditional deep learning models feature engineering is done automatically by the algorithm. Figure 4.6 show the flow of a traditional Machine Learning model. Feature engineering is difficult, time-consuming and it requires a lot of domain knowledge and expertise. Feature learning is easy to accomplish but it requires a lot of data to train the model.

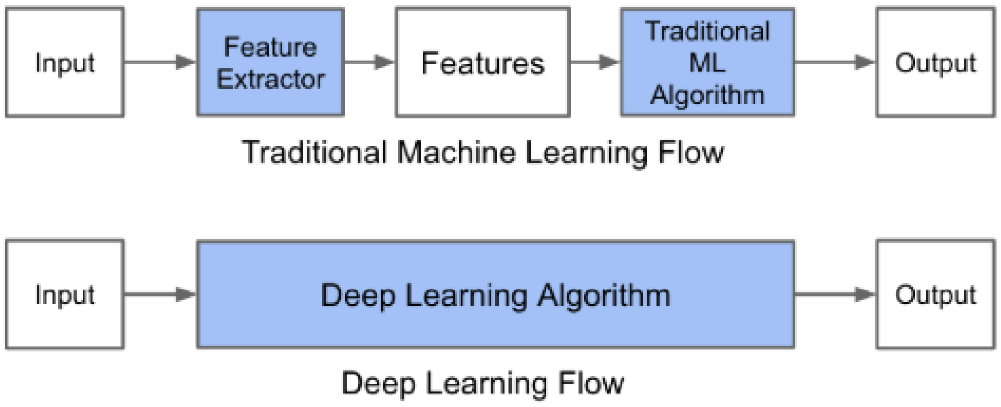


Figure 4.6 Traditional Flow of Deep Learning Algorithm

4.6 Caffe Overview

Caffe is a famous deep learning platform. It was developed by the Berkeley Vision and Learning Center. Caffe is written in C++ and it has Python and MATLAB bindings as well. There are four major steps which needs to be followed for the training a Convolutional Neural Network using Caffe:

- **Data pre-processing:** First of all, image data pre-processing is carried out. Pre-processing involves steps like resizing, histogram equalization and storing images in format that Caffe framework uses. To carry out pre-processing of data we wrote a Python script.

- **Model definition**: In second step, Convolutional Neural Network model is defined by writing CNN configurations for forward training phase in .prototext format file.
- **Solver definition**: The solver makes for model training and optimization. Solver parameters are defined in .prototext format. configuration file
- **Model training:** Netwok training starts by executing Caffe Command described below from the terminal in ubuntu. After training the model, we will get the trained model in a file in .caffemodel format.
- **Model Deployment:** we than use .caffemodel trained model to make predictions of new data. For this purpose, Python Script is written with the help of PyCaffe.

4.7 Data Pre-Processing

First download the data of images from Kaggle to our local machine. Data is zipped so go to command line and unzip it first. Then create lmdb database of these images. For that we wrote python script *create_lmdb.py*. Code snippets of *create_lmdb.py* is given is APPENDIX B (Step1-3).

Create_lmdb.py script performs following functionalities:

- It runs histogram equalization on all images. Histogram equalization adjusts the contrast of all the images.

- Then, resize all training images to a 150x150 dimension which is modeled input for our network.

- Divide the training dataset in two separate categories. One for training and other for validation. About one sixth of all the images in dataset are chosen to be validation images. Training set was used to train the model, and the validation set was used to calculate the accuracy of the model.

- Store the training and validation in two seprate .lmdb databases. train_lmdb for training the model and validatione_lmbd for model evaluation.

Define our Image_transfrom function which takes a colored input images, performs the histogram equalization of the 3 color channels and resizes the image.Code to create lmdb database for input images is shown below. Make_datum takes an input image and its label. Create a datum object with data and label of four dimension.Another common step in preprocessing of image in supervised machine learning is to take the mean of images. Subtracted the mean of image from each input image to make sure that mean of all image is zero. Following command is used to create mean.

4.8 Model Definition

With CNN model finalized and data prepared, now define model in .prototxt file. Caffe provides few popular CNN models such as AlexNet and GoogLeNet. There are also bvlc_reference_caffenet models provided by Caffe. Bvlc_reference_caffenet is basically the definition of AlexNet model. We made changes in bvlc_reference_caffenet file according to our own requirements.

4.9 Solver Definition

Model optimization is done by Solver. It is written in .prototxt format. Define solver's parameters in this file. Solver file computes the accuracy of trained model after every 1000 iteration using validation dataset. The training will run for maximum of 30000 iterations and will take snapshot of trained model after every 5000 iteration. Adjust these values according to different situations. Lr_policy, base_lr, gamma, momentum and weight_decay are some of the hyperparameters that are needed to to make a model converge.

4.10 Model Training

After defining model and solver file in .prototxt format proceed to train model . Command in step 4 of APPANDIX B is used to start training model. Training logs are stored in model_1_train.log file. During the process of training one needs to monitor two values. Loss and Model Accuracy. If training process needs to be stopped at any time all we need to do is to press ctrl+c in command line. Caffe will automatically generate training snapshots with loss and model accuracy. Snapshots are stored in caffe_model_1 folder.

4.11 Learning Curve Plot

A learning curve is a graph which give the accuracy of trained model. Training loss and Test loss is plotted as a function of the number of iterations. These graphs are helpful to visualize the training (or validation) losses and accuracy of trained model. Image 4.2 shows the graph of training and validation loss and accuracy against number of iterations. From this graph it can be sees that this model saturates in performance at about 3000 iterations. Trained model achieved a testing accuracy of 90%.

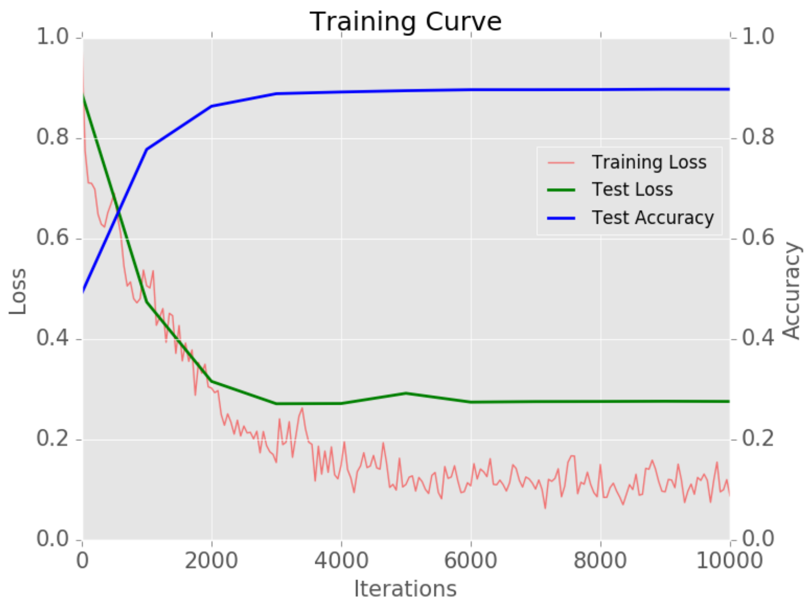


Figure 4.7 Training Evaluation Graph with loss and prediction accuracy

4.12 Predicting New Data

After training of network model is complete it can be used to predict unseen data of images. We write a python code make_prediction_1.py and kept it in the same directory. This code needs four things to run.

- Test images
- Mean of test image
- Model architecture file with SoftMax Function at the end
- Trained model weights

Important parts of the code are listed as step 6 and step 7 in APPENDIX B. Trained model was intiated in python as net and mean image were stored as as mean_array. Deploy trained model needs two things. Trained weights with extension .caffemodel and Model architecture file. And following code reads input image which is test image, applies same pre processing steps as in training phase and computes the probability. Results of training were submitted the predications to Kaggle, it received an accuracy of 0.89691.

CHAPTER 5: FPGA ACCELERATOR DESIGN

In this chapter discusses the design space exploration scheme for implementing Convolutional Neural Networks on FPGA. Section 5.1 and 5.2 discuss about Xilinx Zedboard. Section 5.2 demonstrates why it's important to accelerate the Convolutional Neural Networks. A scheme is provided that analytically models the all operations of CNN and analyze their computation and memory access patterns. Finally, design space on FPGA is explored.

5.1 Xilinx Zynq Zedboard

Xilinx Zynq Zedboard (XC7z020-CLG480-t) is used for mapping of complete CNN. Figure 5.1 shows the diagram of Xilinx Zynq Zedboard. Zedboard is a cost-efficient development environment. It is based on Xilinx Zynq-7000 All programmable System on Chip (AP SOCs). The biggest advantage of Zynq Zedboard is that it has tightly coupled dual core ARM Processing Unit (PS). It is considerably faster than MicroBlaze Soft Processor Core. It has 7-series Programmable logic (PL). It has 512MB DDR3 external memory for PS and 256 MB external Memory for PL. and it supports a 4GB SD Card.

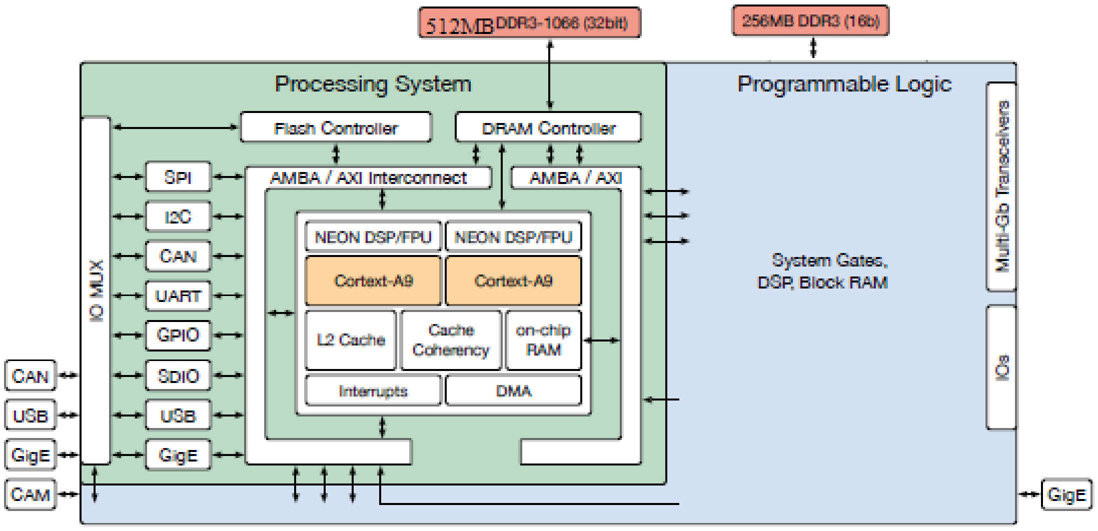


Figure 5.1 Layout of Zynq Zedboard

5.2 High Level Synthesis

Vivado High Level Synthesis (HLS) is a High abstraction Level Synthesis tool for FPGAs. In HLS one can write code and perform debugging in C/C++/SystemC of an FPGA design. An important part of this VIVADO is the VHLS. This compiler understands and converts the higher-level programs into a low-level RTL schematic of the to run this program. HLS automatically optimizes this code to run in parallel as much as possible. It generates schemes for resource allocation and scheduling. Process of synthesis creates RTL

specification. HLS Compiler tests the resultant RTL specification which helps engineers to use the original C/C++/SystemC software as a test-bench.
Synthesis is followed by a process called Co-simulation in VHLS. In this step first of all software codes are executed and all input and outputs are stored. Then the RTL specification is simulated with these stored inputs and outputs. At the end outputs of the software model and output of RTL simulation are matched. If both output match, then o-simulation is successful.
Another useful feature of HLS is that when code is written in C and C++, software specification completely ignores timing and clock cycles in design. So, it is the job of HLS to implement a design in optimized fashion using various compiler directives.
Vivado HLS finds the utilization of all the resources on chip and the maximum clock frequency which a design can utilize after every synthesis. This tool also gives a lot of different analysis views that help us to find the resources which have been allocated to e section of code. It also gives the exact schedules for all loop and function.

5.2.1 Coding Style

UG902, User Guide for High-Level Synthesis [56] for Vivado Design Suite is very good document when working with VHLS. It helps designers to make designs at a level that is higher level of abstraction away from the implementation details regarding RTL. The High-Level Synthesis Blue Book [57] by Mike Fingeroff provides useful insight when designing with BHLS.
Mike Fingeroff presents the importance of the higher-level of abstraction for FPGA designs. But stressing that there is still possibility of resulting in poor RTL schematics when the C/C++ code is not specified well enough. Good style of writing VHLS requires an understanding of the hardware architecture which is to be implemented of an algorithm, it should also reflect not only in C++/C/SystemC code, but also requires an understanding of how HLS behaves. Mike in this document goes to an incredibly low-level of design style like bit level design of registers, muxes and arithmetic operations in the C/C++ code.

5.2.2 Compiler Directives

The high abstraction languages like C/C++ don’t have to capability to make parallelization in respective designs. There are various frameworks which help in explicitly exploiting parallelism in C/C++ programs. They make use of multiple threads which are executed in parallel for example as in CUDA. This allows engineers to mimics the source code with compiler to specify the desired type of parallelism. Now we introduce important ***#pragma HLS*** compiler directives which will be used in proposed Accelerator design.

Interfaces:
Vivado HLS synthesizes functions into various blocs where ach block will get clock and reset ports (ap_clk, ap_rst) by default. Function arguments are automatically result in

different types of RTL ports. Compiler directive for function level interface is shown below.

```
#Pragma HLS INTERFACE <mode> [register] [depth=<D>] port=<P>
```

Interface protocol is subjected at Function level, by applying this pragma on return port of function. Return port is usually marked as handshake signal but it can also be set as nothing, control signal or control chain signal by default. When this pragma is applied it also creates start signal, done signal, ready signal and idle signal which let the blocks in communicating for data transfers. Which interface protocol to apply on function arguments also depends on what kind of argument is under discussion. There are various modes like stream, memory mapped and high-speed burst mode for data transfer or simple memory ports.

Loop Unrolling:
Loop unrolling pragma is shown below. This pragma when applied on certain loop asks the compiler to unroll all the loop iterations. Loop unrolling can be either complete or partial with factor of N. All operations are scheduled as quickly as they are ready for execution by VHLS. Parallel execution of these unrolled loop iterations than carries out. Data dependency and external memory bandwidth usually limits the amount of unrolling in loops. For complete unrolling of loop external bounds must be known already.

```
#pragma HLS UNROLL [factor=<N>]
```

Loop Pipelining:
Another very important optimization directives in HLS is loop pipelining. This pragma pipelines the section of code in which it is placed, and all that is in below this section. HLS automatically tries a pipeline code section with an initiation interval of one. This means that after pipeline every new clock cycles a new input can be taken.

```
#pragma HLS PIPELINE [II=<N>]
```

Resource Specification:
This pragma applies a specific resource on any given variable (var) which is being implemented in RTL. This is important when we wish to use a special type of memory for example for an array like single port or dual port.

```
#pragma HLS RESOURCE variable=<var> core=<string> [latency=<N>]
```

Function Inlining:

This pragma is shown below. It forces a specific function which called many times to be inlined into all its callers. This makes multiple same hardware of said function. So, it results in increased resource utilization which is turn increases throughput. But eventually it comes down to tradeoff between the two area and speed. Vivado HLS more often than not automatically inlines functions. This pragma help increase design throughput at cost of area consumption.

```
#pragma HLS INLINE
```

5.3 CNN: Computational Analysis

Convolutional Neural Networks are computation intensive algorithms. They involve huge memory operations and perform millions of computations. Trained networks are algorithmically simple to execute. Chapter.2 presented a discussion about different types of layers in Convolutional Neural Networks. Compute intensive nature of those CNN architectures is discussed and analyzed. Later, this analysis is used to implement complete convolutional Neural Network on Zynq Zedboard.

Convolutional Layer:

Study of convolutional neural network suggests that 90% of the time of forward execution of a CNN is spent on convolutions [18]. Speeding up the process of convolution accelerates entire working of convolutional neural network. Figure.5.1 shows depiction of a Convolutional Layer. Convolutional Layer has input feature maps of dimension Rin x Cin x N, and output feature maps of dimension Rout x Cout x M, where N denotes input channels and M denotes output channels. Equation.2.2 and Equation.2.3 give the dimensions of output feature maps depending on the values of size of kernel window (K), padding (P) and stride of kernel window (S). To execute a convolutional layer RinxCinxN input pixel and NxMxKxK weights values are read from DRAM. In each iteration of Convolutional Layer one addition and one multiplication is performed. So, in each in a convolutional layer 2xRxCxNxMxKxK operation are performed which accounts for RxCxNxMxKxK Multiply-Accumulates (MACCs). R x C x M output pixels are written back to DRAM.

$$\begin{aligned}\boldsymbol{OFM}[\boldsymbol{M}][\boldsymbol{R}][\boldsymbol{C}] &= \sum_{\boldsymbol{0}}^{\boldsymbol{N}} \sum_{-\boldsymbol{k}/\boldsymbol{2}}^{\boldsymbol{k}/\boldsymbol{2}} \sum_{-\boldsymbol{k}/\boldsymbol{2}}^{\boldsymbol{k}/\boldsymbol{2}} (\boldsymbol{IFM}[\boldsymbol{S}*\boldsymbol{R}+\boldsymbol{i}][\boldsymbol{S}*\boldsymbol{C}+\boldsymbol{j}][\boldsymbol{N}] \\ &* \boldsymbol{Weight}[\boldsymbol{M}][\boldsymbol{N}][\boldsymbol{i}][\boldsymbol{j}]) \quad (\mathbf{5.1})\end{aligned}$$

Table 4.2 shows the network we want to implement. Equation 5.1 shows convolution operation of Figure.5.1To perform convolution one 67,500 input pixels are read from DRAM. 864 weight values are read from memory and a total of 58 Million Convolutions are performed. And then, 2,160,000 pixels are written back to DRAM. This provides the foundation of our accelerator design. Accelerator design in discussed in section 5.7.

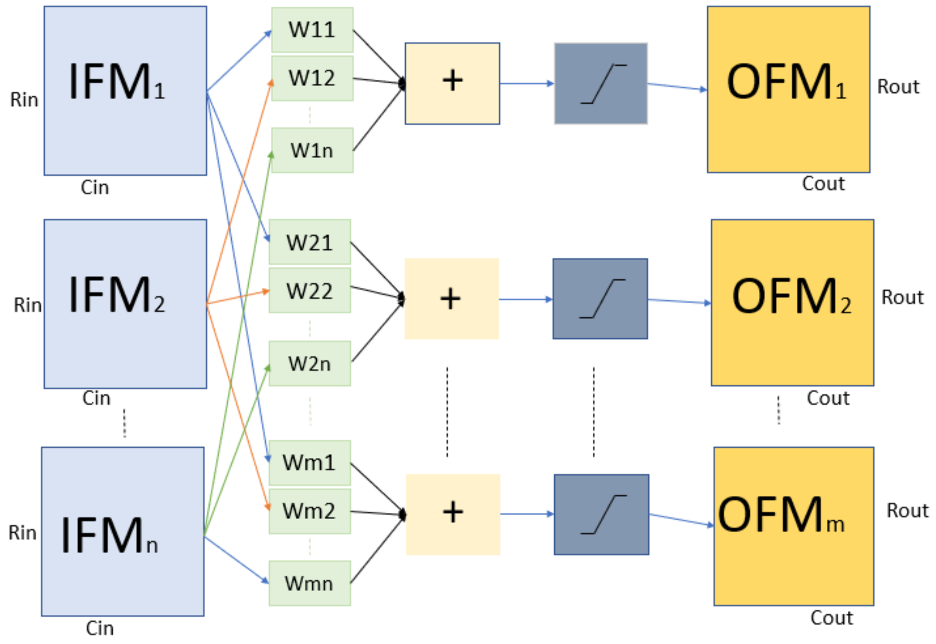


Figure 5.2 Pictorial Description of Convolutional Layer

Pooling Layer:

Convolutional Layer is usually followed by a Pooling / Sub-Sampling Layer. Pooling Layer down samples input feature map. It provides scale and distortion invariance. Figure 5.2 shows the pictorial depiction of a pooling layer. Dimension of output image depends upon the pooling window size (P). pooling window size is usually 2x2 or 3x3. In each operation of Pooling Layer RxCxM input pixels are read and R/P x C/P x M pixels are written back. Where P is 2 or 3 depending on the size of pooling window. In Pooling layer RxCxMxMxPxP Boolean Operations are performed.

Fully Connected Layers:

Fully Connected layers are ANN layers. These layers have been explained previously. They are nothing more than dot product. Equation 2.1 gives the relationship of input neurons to output neurons. If there are N input neurons and M output neurons. A total of N input neurons are read from memory NxM MACCs are performed and M output neurons are written back to memory.

5.4 Software and Hardware Partitioning

From previous discussion it is quite clear that Convolutional Neural Networks are computationally expensive algorithms. They have huge amount of memory accesses and they perform millions of operations. Of all the layers Convolutional Layers are most expensive layer. about 90% of the time of convolution is spent in Convolutional layer. Speeding up the process of convolution result in speed up achieved for entire working of Convolutional Neural Network. It is proposed to partition CNN algorithm into hardware part and Software Parts. Convolutions are computationally the most expensive part of algorithm hence they are mapped on custom hardware. Where one can exploit parallelism in workload of CNN and also design our custom DRAM controller to efficiently manage off-chip memory traffic. Other tasks such as Pooling Operation, Dot Product in Fully Connected layer and SoftMax layers are not computationally so expensive. Hence, they are mapped on Software in a host computer.

Some of the key features of our Hardware Software Partitioning Scheme are as follows.

Software part of algorithm deals with operations that consume less time to complete. Dot product or pooling or mathematically complex tasks such SoftMax operations. Which can not show any significant improvement on Custom hardware design. In Software part following tasks are performed

- Define a network architecture as shown in Table.4.2.
- Model forward propagation of network.
- Define and allocate memory for keeping weights and feature maps. Malloc () function was used to create dynamic memory allocation for input feature maps and output feature maps and weights.
- Read weights of Caffe Model which were then converted to .bin format by a python script. Load these weights into memory allocated for weights previously.
- Read Input image for testing of network. Load this image into memory allocated previously.
- Test input image by making forward propagation

Hardware part of algorithm purely focuses on the Convolution. Convolutions being the most time-consuming part of CNN. A custom Accelerator design on FPGA is presented which takes into account memory access pattern of CNN and different levels of parallelism that exists in CNN. Analytically modeling was carried out to find the best possible solution which gives the highest performance. Details of Hardware Accelerator will be discussed in next section

5.5 Accelerator Design

This section discusses the custom Accelerator design for carrying out convolution process.

5.5.1 Different Levels of Parallelisms in CNN

In CNN, there are different several sources of parallelism. To find an accelerator design with best possible performance one needs to exploit these parallelisms.

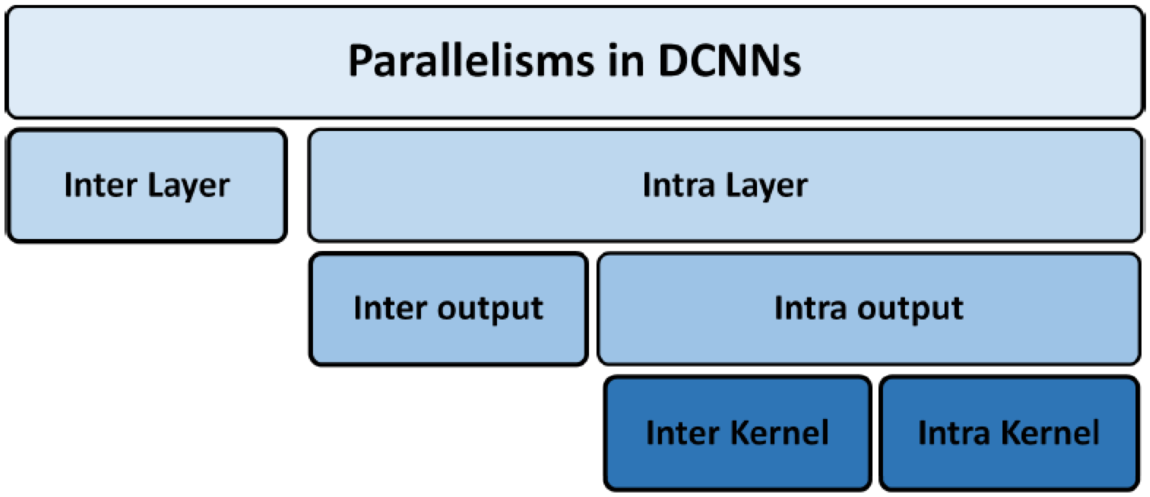


Figure 5.3 Different Levels of Parallelism in Convolution Layer workload

Inter Layer Parallelism

In Convolutional Neural Network

$$\forall\, \boldsymbol{layer} \in \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \ldots, \boldsymbol{L}\}: \boldsymbol{IFM}(\boldsymbol{layer} + \mathbf{1}) = \boldsymbol{OFM}(\boldsymbol{layer})$$

is dependency between two subsequent layers; hence, so it is not possible to execute two subsequent layers in parallel. Additionally, since state of art CNNs are huge in size so it is also not feasible to pipeline the operations of all CNN layers. Even for small sized CNN, pipelining doesn’t usually give good performance.

Inter Output Parallelism

Output feature maps in a Convolutional Neural Network are totally independent of each other. In theory all of these output feature maps can be executed in parallel. To achieve inter output parallelism equation.5.1 is computed for different values of M in parallel.

Inter Kernel Parallelism
On pixel in output feature map is result of K^2 convolutions. And these K^2convolutions are independent of each output channel, this presents the possibility to compute all K^2 convolutions at once. This provides another source of data level parallelism which can be achieved. In order to exploit inter kernel parallelism, Equation.5.1 is calculated for different values of R and C.
Intra Kernel Parallelism
There is a vast amount of parallelism in each K^2 convolution. A convolution is one MACC operation which consists of one multiplication and one addition. Weight kernel size of 3 would produce 9 MACC operations. All of these multiplication between a kernel and a pixel in an input feature map is independent from another multiplication, all of them can be theoretically computed in parallel.

If an FPGA chip has unlimited area, on chip memory (BRAM) and external memory bandwidth. all of these sources of parallelism could be exploited to speed up the operation of convolution in a CNN. But practically it is not possible. Therefore, we find the optimal combination of different level of parallelism which both minimizes the execution time and satisfies the constraints on target chip.

5.5.2 The Roofline Model

Computation and communication are two constraints to find system performance for optimization. Performance of an algorithm is restricted either by external memory-accesses or computation capability. In [55], a Roofline performance model was developed which relates system performance to off-chip memory traffic and gives peak performance of hardware platform to be used. Equation.5.2 gives the performance of an algorithm on a certain platform according to Roofline Model.

$$Peak\ Peformance = \min\{Computationl\ Roof, CTC\ Ratio\ x\ BW\ of\ IO\} \quad (5.2)$$

Floating points per second (GFLOP/sec) is used as metric to evaluate system throughput. Floating Points per Second can be no higher than two terms in equation. 5.2. first term is peak GFLOPS provided by all available computational resources in a system which is also called computational roof.

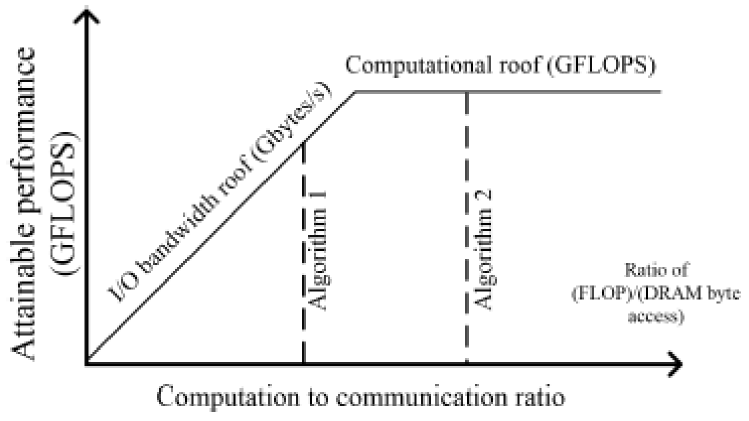


Figure 5.5 Basis of Roofline Model

Second term features off-chip memory traffic which operations per DRAM Byte Access called CTC (Computation to Communication) ratio.
Figure 5.5 provides a visualization of roofline model. Algorithm 2 has higher performance as it has higher computation to communication ratio. It means that algorithm 2 has better reusability of data. Algorithm is restricted by external memory bandwidth. So, it can not perform higher than Algorithm 2.

5.5.3 Computation Model

This section presents design of Accelerator based on all the previous considerations. We implement CNN on Zynq Zedboard which has dual core ARM Processing unit (PS) and 7-Series Programmable Logic (PL). We design our Convolution Accelerator on Zynq-PL. Whereas other operations like pooling, dot product and SoftMax have already been assigned to Zynq-PS. Figure 5.6 gives a depiction of our accelerator design on Zynq-PL.

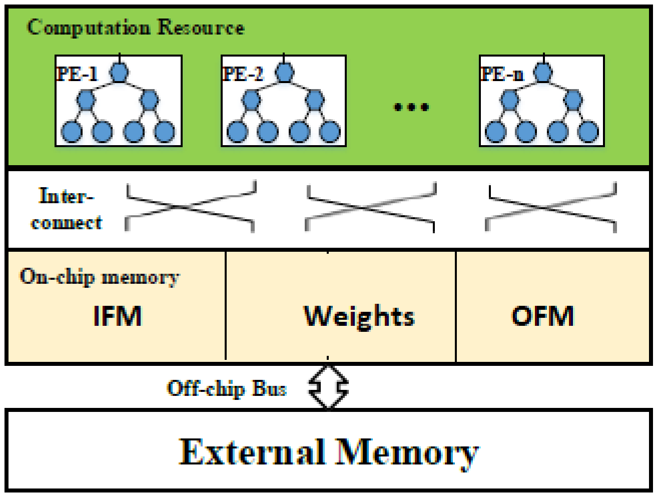
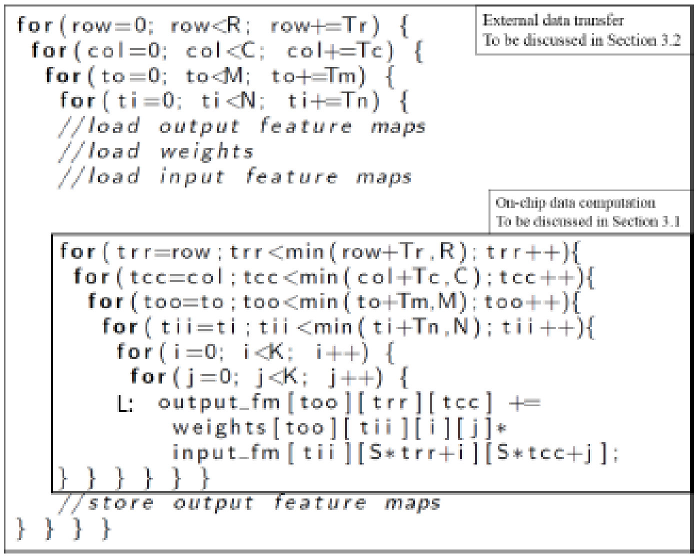


Figure 5.6 Proposed Accelerator Design

Our Accelerator primarily consists of Processing Elements (PE) which are responsible of performing convolutions, on chip BRAM buffers for storing input feature maps and output feature maps and weights. It consists of on chip interconnects which are responsible for dataflows through the Processing Elements.

There are various design challenges which obstacle the performance of Accelerator design. First of all, Loop Tiling is important to fit small portion of input data, output data and weight data. One needs to make sure choose correct tiling scheme. An improper tiling scheme reduces the performance of system. Second, it is important to carefully consider the on-chip buffers, processing elements and interconnects between them to have a better reusability of on-chip data. Poor reusability of on chip data degrades the performance of Accelerator. Third and the most important is that throughput of processing elements should match the off-chip memory bandwidth provided by Zynq Zedboard platform.

Figure 5.5 gives the algorithm structure for data transfer to accelerator and on-chip computation of convolutions. As discussed previously, first apply loop tiling or point loop to fit the small portion of data on-chip. Table 4.2 shows that except the values of kernel window in all convolutional layers all the values change. So, restrict the values of R, C, M and N to TR, TC, TM, and TN. This makes a tile size tuple of <TM, TN, TR, TC>. In subsequent section we discuss how to make loop tiles.



External data transfer
To be discussed in Section 3.2

```
for(row=0; row<R; row+=Tr) {
 for(col=0; col<C; col+=Tc) {
  for(to=0; to<M; to+=Tm) {
   for(ti=0; ti<N; ti+=Tn) {
   //load output feature maps
   //load weights
   //load input feature maps
```

On-chip data computation
To be discussed in Section 3.1

```
for(trr=row; trr<min(row+Tr,R); trr++){
 for(tcc=col; tcc<min(col+Tc,C); tcc++){
  for(too=to; too<min(to+Tm,M); too++){
   for(tii=ti; tii<min(ti+Tn,N); tii++){
    for(i=0; i<K; i++) {
     for(j=0; j<K; j++) {
    L:  output_fm[too][trr][tcc] +=
           weights[too][tii][i][j]*
           input_fm[tii][S*trr+i][S*tcc+j];
} } } } } }
   //store output feature maps
} } } }
```

Figure 5.5 Pseudo Code for Data Transfer and On-Chip Computation of Convolution

Loop Tiling

Pseudo Code of Convolutional Layer is shown in figure.5.5. multi-dimensional convolution which exists in CNNs is nothing but a nested loop presentation of algorithm in Equation.5.1. this nested loop presentation of algorithm provides the basis of computation for our FPGA Top Function.

But FPGA designs strictly follow a certain pattern based on clock cycles. An FPGA top function operates for a specific number of clock cycles. But Table.4.2 shows that no parameters are same for layers except size of kernel window. So, it is not possible to efficiently map all convolutional layers of CNN on FPGA with the parameters that are varying all the time. Besides, limited amount of on-chip memory makes it impossible to load the all parameters and weights of a CNN layer on FPGA concurrently. So, using tiling to load specific set of data on FPGA BRAM. Proposed FPGA top function runs for a specific number of rows, columns, input channels and output channels and kernel. Our tiling scheme is inspired by the work of [8] [9]. Goal of our accelerator design is to deliver a complete CNN implementation our choice of tiles size differs from [8] [9] in a sense that we use lowest value of input channels and output channels, rows and columns across all layer. From Table. 4.2 we can see that lowest value of input channels, output channels, rows, columns are 3,32,18,18 respectively. Experimentally it was found that rows and columns of tile size 18 doesn't give optimum performance on Software. Figure 3 gives the graph of latency of convolutional layer against different tile sizes of rows and columns. Tile Size of 30 gives the best performance across all convolutional layers. So, a tile size tuple of <3,32,30,30> was chosen.

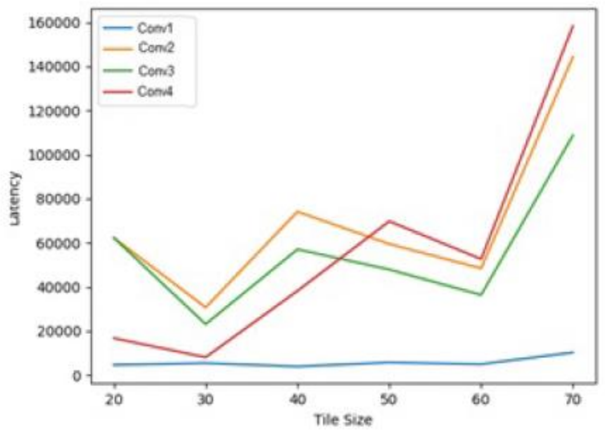


Figure 5.4 Performance Analysis of Different Tiles size of Rows and Columns

Optimization of Computation Engine

This section presents computation engine optimization. Objective of these optimization is to take full advantage of loop unrolling and loop pipelining. To perform loop unrolling and loop pipelining it was assumed that input data, weight data and output data was already buffered on chip.

Loop Unrolling helps us utilize huge amount of computation resources that are available in an FPGA Platform. Unrolling CNN Convolutional layer along different dimension results in different implementation models. loop unrolling along any dimension is affected by the data dependencies and this eventually determines the complexity of design. For a given array data sharing relation for a certain loop dimension can be categorized in three ways.

Irrelevant: If a loop iterator does not appear in any access function of an array ARR, then this loop dimension is irrelevant to ARR.

Independent: If a loop iterator is totally separable along any dimension of an array ARR than it is independent to ARR.

Dependent: If a loop iterator is not totally separable then, it is dependent to array ARR.

Figure 5.5 gives the implementation detail of data sharing relationship between a loop iterator and array. Irrelevant iteration results in broadcasting connectivity. Independent titrator creates direct connections while dependent iterator results in complex connectivity.

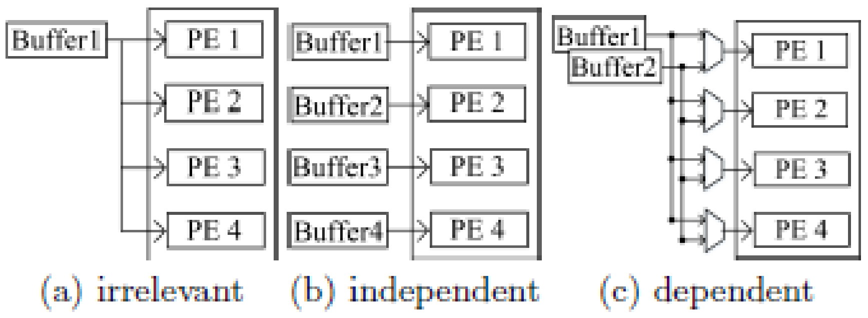
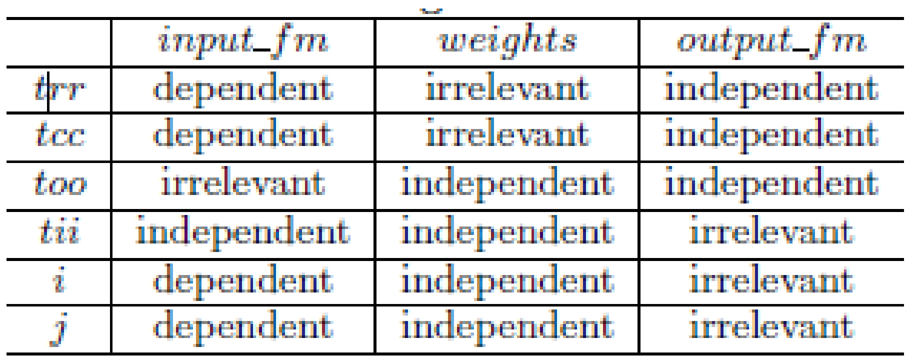


Figure 5.5 Hardware Implementation for different data sharing relation

Figure 5.6 gives the detail of dependencies of loop iterators with respect to the arrays input feature map, output feature map and weights. As we can see that loop dimension too and tii don't have any dependencies along any array. Unrolling these two dimensions generates the best possible result.

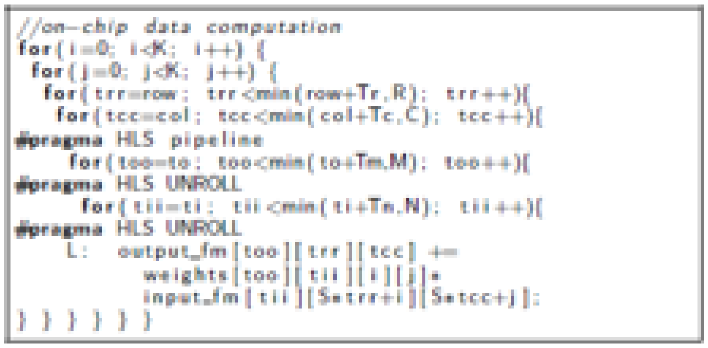


	$input\_fm$	$weights$	$output\_fm$
trr	dependent	irrelevant	independent
tcc	dependent	irrelevant	independent
too	irrelevant	independent	independent
tii	independent	independent	irrelevant
i	dependent	independent	irrelevant
j	dependent	independent	irrelevant

Figure 5.6 Dependency analysis of loop iterator along different arrays

Loop Pipelining: Loop pipelining is the most important pragma in High Level Synthesis (HLS) to improve the system throughput. It overlaps the execution of operations for different loop iterations. Data dependencies and resource constraint limit the performance achieved by loop pipelining. Loop carried Dependency prevents the loop from unrolling completely. It also stalls the pipeline process.

Figure 5.7 gives the detail of optimizations. Loop unrolling was carried out along two loop iterations, too and tii to make custom computing engine. And finally, loop pipelining pragma was applied on top of these two loop iterations. This design gave best throughput. Resultant computing engine is shown on the right hand side. Now pixels of Tn input feature maps combine together to form Tm output feature maps.



```
//on-chip data computation
for(i=0; i<K; i++) {
 for(j=0; j<K; j++) {
  for(trr=row; trr<min(row+Tr,R); trr++){
   for(tcc=col; tcc<min(col+Tc,C); tcc++){
#pragma HLS pipeline
    for(too=to; too<min(to+Tm,M); too++){
#pragma HLS UNROLL
     for(tii=ti; tii<min(ti+Tn,N); tii++){
#pragma HLS UNROLL
    L:  output_fm[too][trr][tcc] +=
          weights[too][tii][i][j]*
          input_fm[tii][S*trr+i][S*tcc+j];
} } } } } }
```

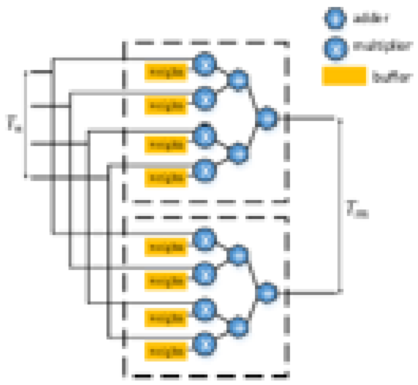
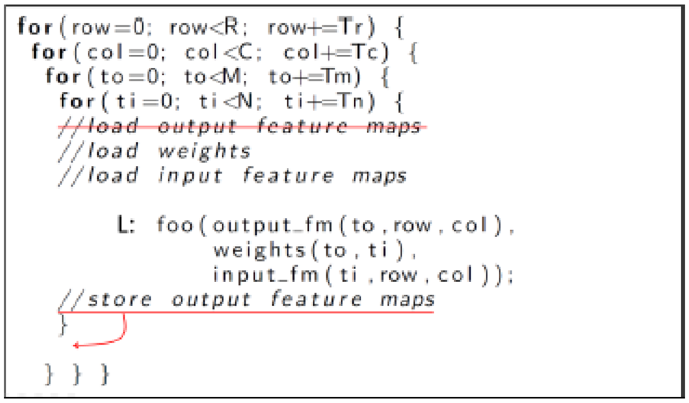


Figure 5.7 Computation engine optimizations and resultant computation engine

Optimization of External Memory Accesses

Previous section presented a discussion on how to derive different implementations of computational engine having different computational roofs. A design which has high roof of computation does not necessarily mean higher performance due to memory bandwidth constraint. All input data (IFM, OFM, Weights) are already buffered to computation engine on-chip. Figure 5.8 explains the external memory access operations of a Convolutional layer. Input feature maps, output feature maps and weights are loaded before the computation engine starts working and the resultant output feature maps are written back to main memory.



```
for(row=0; row<R; row+=Tr) {
 for(col=0; col<C; col+=Tc) {
  for(to=0; to<M; to+=Tm) {
   for(ti=0; ti<N; ti+=Tn) {
   ~~//load output feature maps~~
   //load weights
   //load input feature maps

       L: foo(output_fm(to,row,col),
              weights(to,ti),
              input_fm(ti,row,col));
   //store output feature maps
   }

  } } }
```

Figure 5.8 Memory Access of a Convolutional Layer

As figure 5.8 shows that innermost loop dimension ti is completely irrelevant to array output feature map. Hence there is a redundant memory operation for array output feature map. To optimize this design, this operation was promoted to outer loop by one step. Operation of reading output feature map is performed in loop iterator to. This memory promotion scheme reduces the external memory access for Accelerator and also improves computation to communication ratio.

Accelerator Design Flow

Nested-loop algorithm presented in Figure.3 provides the basis for FPGA Accelerator design. Each of these loops in FPGA accelerator runs a specific number of times depending on tile-size tuple. At the start of layer execution, Zynq-PS points to the reference of location of data and accelerator starts cashing the data from that particular location, performs the convolutions and stores output data back to memory at a particular location pointed to by Zynq-PS. For achieving a good throughput of Accelerator, two things need to be ensured in design. On chip caching of input data and output data and parallelization between the execution of different convolution operations. Line buffers were introduced for input data, output data and weights. Line buffers on each port ensure continuity of dataflows. There are different levels of parallelization dis- cussed in work [8] [9]. Proposed methodology achieves inter-output parallelism where pixels of Tn input feature maps combine together to form Tm output feature maps. One critical part of proposed Accelerator is DRAM Controller. DRAM Controller is designed taking into consideration memory access patterns of CNN.

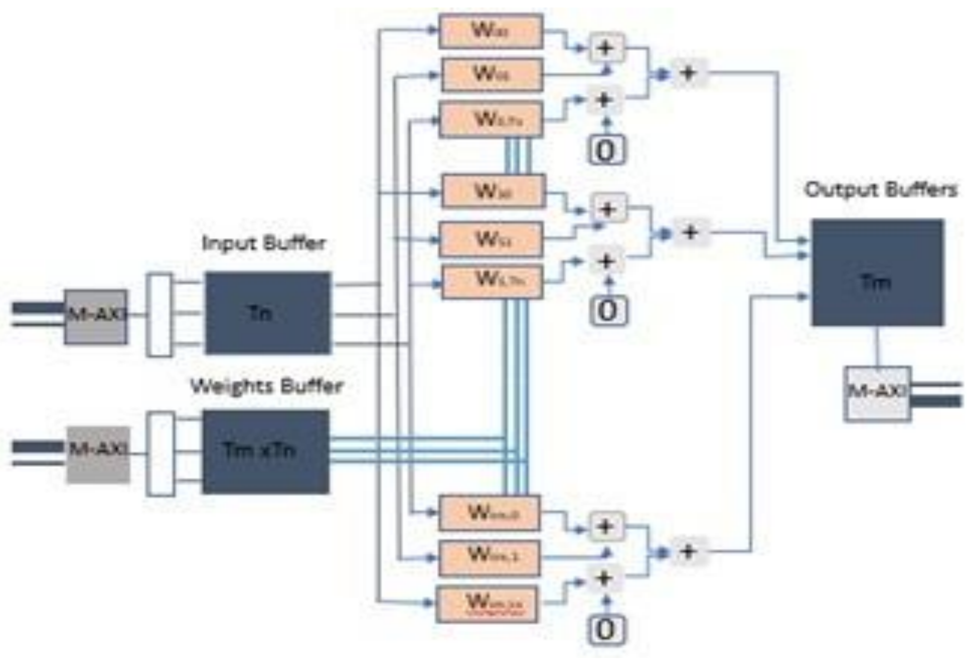


Figure 5.8 Accelerator Design

All the data is read from and written to DRAM by M-AXI interface protocol using Burst Mode with a Burst length of 128. Loop pipelining and loop unrolling are two pragmas that are used in HLS which helps in achieving high throughput of design. HLS Unroll brings parallelization into design. HLS Pipeline gives high throughput. Figure 4 shows the architecture of proposed Accelerator design on Zynq PL

5.6 Experimental Setup:

Accelerator design was implemented in Xilinx Vivado High Level Synthesis tool (2018.1). HLS transforms codes written in C/C++/SystemC to RTL with the help of HLS-defined Pragmas. Proposed Accelerator design was written in C++ language. Working of Accelerator Design was than tested with Test Bench written in C++ language. Timing analysis and C/RTL Co-Simulation was then carried out to further check the validity of design. Pre-Synthesis Reports help in design space exploration and to evaluate the performance of a design. Resulting RTL design was than exported to Vivado for Synthesis. Synthesized Design was implemented using SDK.

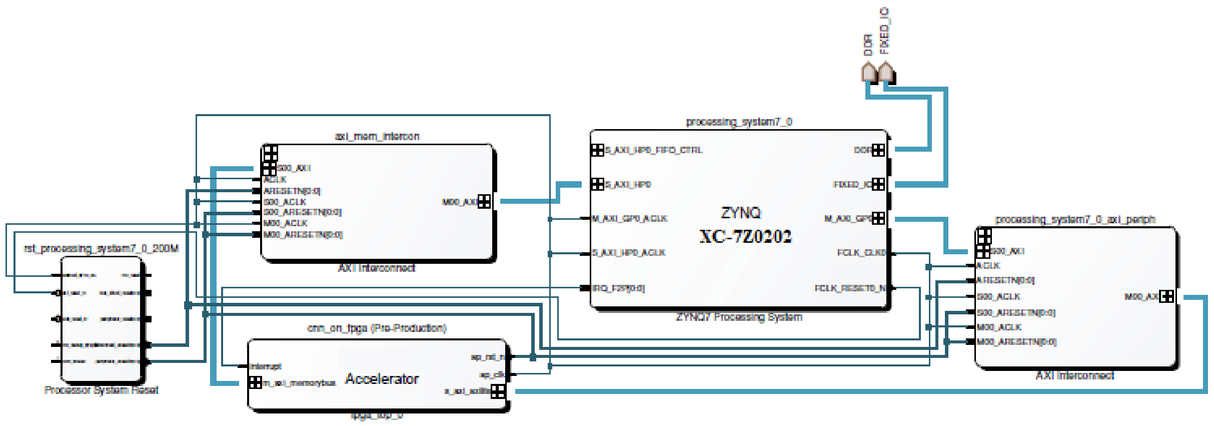


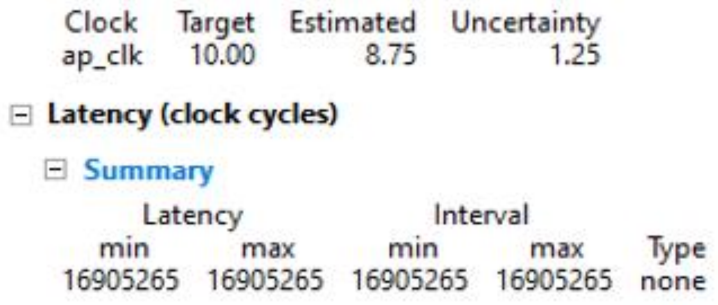
Figure 5.8 Accelerator Design Schematic on Xilinx Vivado

CHAPTER 6: RESULTS

First target of our project was to design a proof of concept FPGA Accelerator for CNN applications. Our second target is to achieve maximum possible throughput on the given platform. This platform in turn gives efficiency in power consumption. Implementation details of proposed design were discussed completely in last chapter. This chapter gives the results of simulations in Vivado and discuss throughput maximization step by step. Latency of design was used as parameter for measuring the throughput of our accelerator. These latency values have been noted for tiled Convolution. In each tiled convolution 777600 MACC Operations were performed.

Step 1: Base CNN Code

In first step base CNN code was implemented for tiled convolution on FPGA. It resulted in a latency of 16905265 clock cycles which approximates to 147 milli seconds to perform 777600 MACCs. This design could run up to 115MHz. Fig.5.1 shows the results of our simulation.



Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.75	1.25

⊟ Latency (clock cycles)

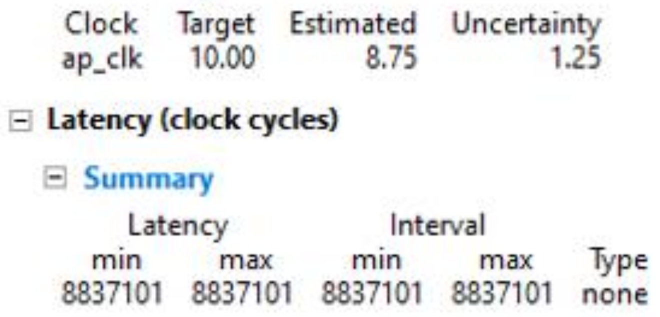
⊟ Summary

Latency		Interval		
min	max	min	max	Type
16905265	16905265	16905265	16905265	none

Figure 6.1 Result 1

Step 2: Base CNN Code Implementation Pipeline and Unroll

In Second step base CNN code was implemented for tiled convolution on FPGA with loop pipeline and unroll. It resulted in a latency of 8837101 clock cycles which approximates to 77 milli seconds to perform 777600 MACCs. This design could run up to 120MHz. Fig 5.2 shows the results of our simulation.



Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.75	1.25

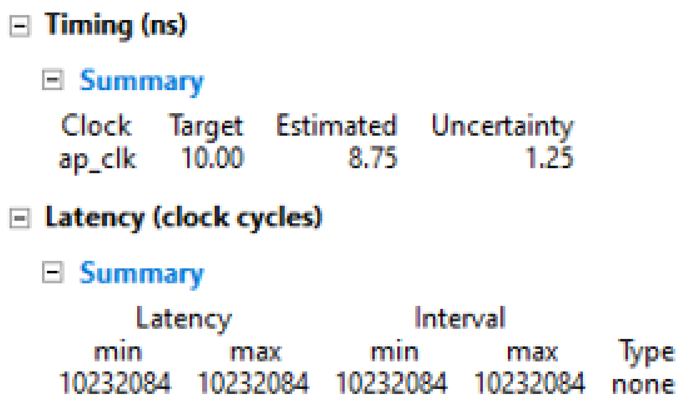
⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Type
8837101	8837101	8837101	8837101	none

Figure 6.2 Result 2

Step 3: Expansion of Design Space Exploration Without any optimization
In third step operations of tiled convolution were expanded with introduction of on chip buffers. No optimization directive was used. It resulted in a latency of 10232084 clock cycles which approximates to 89 milli seconds to perform 777600 MACCs. This design could run up to 120MHz. Figure 5.3 shows the results of our simulation.


Timing (ns)

Summary

```
Clock    Target   Estimated   Uncertainty
ap_clk   10.00         8.75          1.25
```

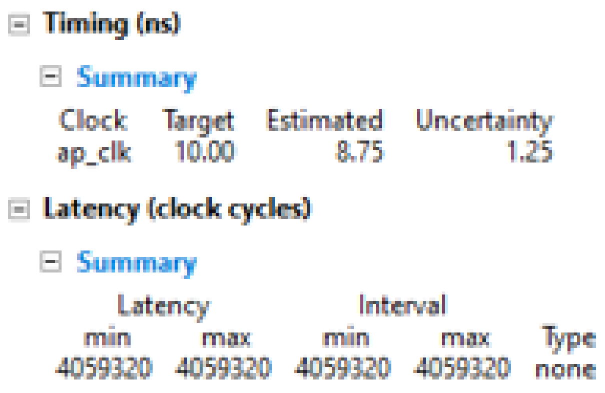
Latency (clock cycles)

Summary

```
        Latency                 Interval
   min        max          min        max      Type
10232084   10232084    10232084   10232084     none
```

Figure 6.3 Result 3

Step 4: Expansion of Design Space Exploration with Optimization
Next with expanded design space exploration scheme for operations of tiled convolution, Loop Unroll and Loop Pipeline optimization directives were introduced. This resulted in a latency of 4059320 clock cycles which approximates to 35 milli seconds to perform 777600 MACCs. This design could run up to 120MHz. Figure 5.4 shows the results of simulation.


Timing (ns)

Summary

```
Clock    Target   Estimated   Uncertainty
ap_clk   10.00         8.75          1.25
```

Latency (clock cycles)

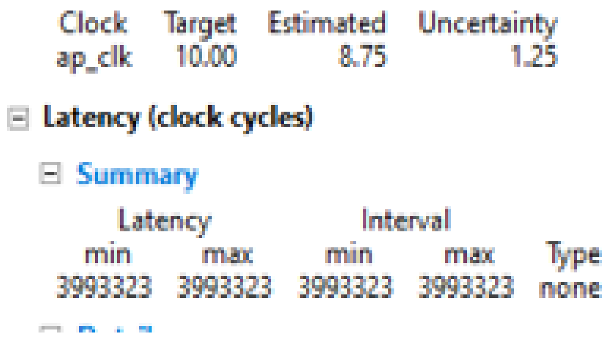
Summary

```
      Latency               Interval
  min       max         min       max      Type
4059320   4059320   4059320   4059320      none
```

Figure 6.4 Result 4

Step 5: Expansion of Design Space Exploration with further Optimization

Next further expansion of design space exploration scheme was carried out for operations of tiled convolution. Previously, Loop Unroll and Loop Pipeline optimization directives were introduced. Now, dual ported BRAM buffers were used for on chip data of input feature maps, output feature maps and weights. This resulted in a latency of 3993323 clock cycles which approximates to 34 milli seconds to perform 777600 MACCs. This design could run up to 120MHz. Figure 5.5 shows the results of our simulation.



Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.75	1.25

Latency (clock cycles)

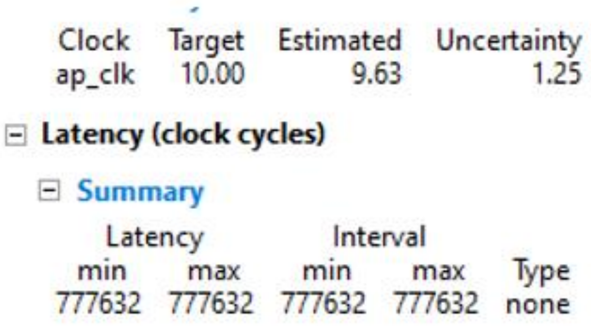
Summary

Latency		Interval		
min	max	min	max	Type
3993323	3993323	3993323	3993323	none

Figure 6.5 Result 5

Step 6: Removing Data Dependency for Differ Loop iterations.

Unlike previous steps a modification was made in design to take into consideration reusability of on chip data. Loop transformation was carried out to increase computation to communication ratio. In this scheme all our previous optimization directives like Loop Unroll, Loop Pipeline optimization directives and dual ported BRAM for on chip data of input feature maps, output feature maps and weights were introduced. It resulted in a latency of 777632 clock cycles which approximates to 6 milli seconds to perform 777600 MACCs. This design could run up to 100MHz. Figure 5.5 shows the results of our simulation.



Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.63	1.25

Latency (clock cycles)

Summary

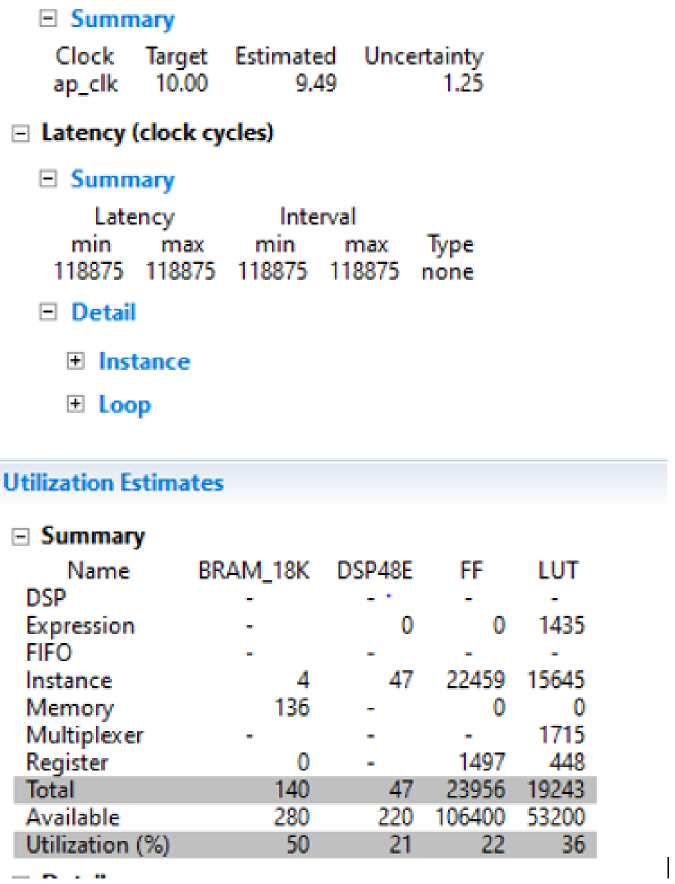
Latency		Interval		
min	max	min	max	Type
777632	777632	777632	777632	none

Figure 6.5 Result 5

Step 7: Improving dataflows in Accelerator design

In this step loop transformation was carried out where loop for input channel was demoted outward and loop for output channel was promoted inward. This optimization improved the flow of data from on chip buffers to convolution computing engine. Improved dataflow resulted in higher throughput of accelerator. This experiment was carried with same tile size. In this scheme all our previous optimization directives like Loop Unroll, Loop Pipeline optimization directives and dual ported BRAM for on chip data of input feature maps, output feature maps and weights were introduced. It resulted in a latency of 118875 clock cycles which approximates to 1 micro seconds to perform 777600 MACCs. This design could run up to 100MHz. Figure 5.5 shows the results of our simulation.

In all previous results there was always under utilization of logic resources. This was because of data dependancies in design. This design improved the resource utilization.



Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.49	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
118875	118875	118875	118875	none

Detail

Instance

Loop

Utilization Estimates

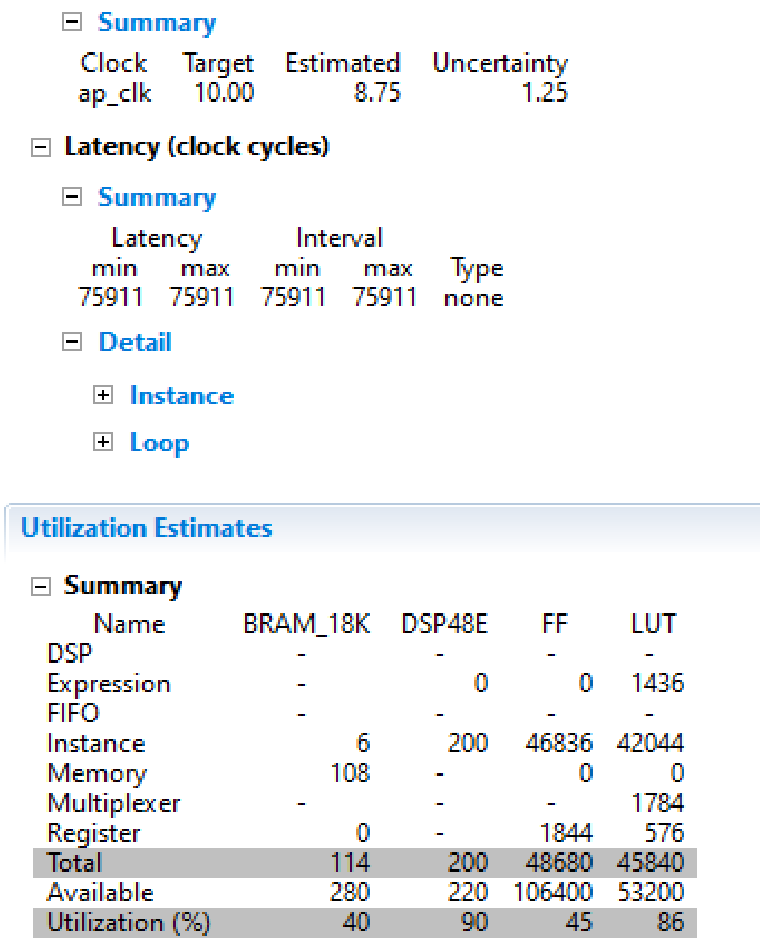
Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	0	0	1435
FIFO	-	-	-	-
Instance	4	47	22459	15645
Memory	136	-	0	0
Multiplexer	-	-	-	1715
Register	0	-	1497	448
Total	140	47	23956	19243
Available	280	220	106400	53200
Utilization (%)	50	21	22	36

Figure 6.6 Result 6

Step 8: New Architectural Template

In this step after removing data dependencies and design of accelerator was reverted to original pseudo code shape. This experiment was carried with a bigger tile size to improve the underutilization of logic resources. With the improvement of dataflow this design was capable of maximum parallel execution. In this scheme all our previous optimization directives like Loop Unroll, Loop Pipeline optimization directives and dual ported BRAM for on chip data of input feature maps, output feature maps and weights were introduced. It resulted in a latency of 75911 clock cycles which approximates to 0.6 milli seconds to perform **1,782,000** MACCs. This design could run up to 115MHz. Figure 5.5 shows the results of our simulation.



Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.75	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
75911	75911	75911	75911	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	0	0	1436
FIFO	-	-	-	-
Instance	6	200	46836	42044
Memory	108	-	0	0
Multiplexer	-	-	-	1784
Register	0	-	1844	576
Total	114	200	48680	45840
Available	280	220	106400	53200
Utilization (%)	40	90	45	86

Figure 6.7 Result 7

Resource Utilization:

Table 6.1 shows resource utilization on Zynq-PL for final and most optimized design.

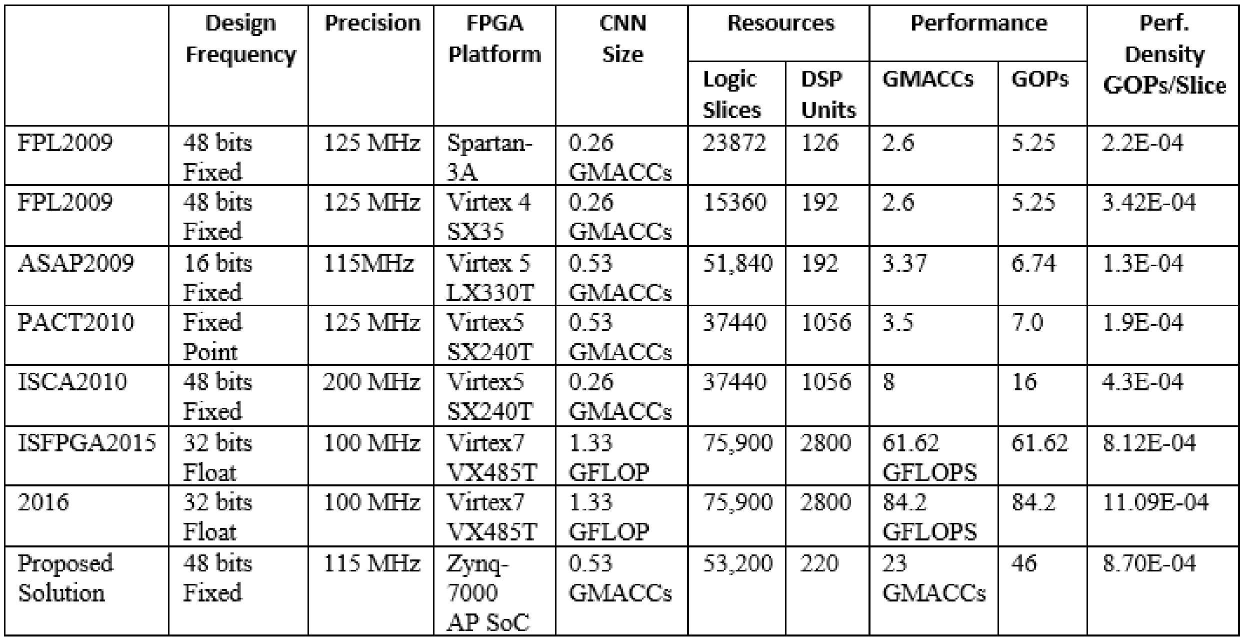
Table 6.1 Resource Utilization on Zynq PL

	BRAM_18K	**DSP48E**	**FF**	**LUT**
Total	114	200	48680	45840
Available	280	220	106400	53200
Utilization (%)	40	90	45	

Benchmark Comparison:

Table 6.2 shows comparison of most optimized design to other state of art implementations. We achieve a performance density of 8.70E-04 which is better than ISFPGA and slightly less than 2016 implementation. This mostly comes from lack of logic resources that we had at our proposal.

Table 6.2 Result Comparison



	Design Frequency	Precision	FPGA Platform	CNN Size	Resources		Performance		Perf. Density GOPs/Slice
					Logic Slices	DSP Units	GMACCs	GOPs	
FPL2009	48 bits Fixed	125 MHz	Spartan-3A	0.26 GMACCs	23872	126	2.6	5.25	2.2E-04
FPL2009	48 bits Fixed	125 MHz	Virtex 4 SX35	0.26 GMACCs	15360	192	2.6	5.25	3.42E-04
ASAP2009	16 bits Fixed	115MHz	Virtex 5 LX330T	0.53 GMACCs	51,840	192	3.37	6.74	1.3E-04
PACT2010	Fixed Point	125 MHz	Virtex5 SX240T	0.53 GMACCs	37440	1056	3.5	7.0	1.9E-04
ISCA2010	48 bits Fixed	200 MHz	Virtex5 SX240T	0.26 GMACCs	37440	1056	8	16	4.3E-04
ISFPGA2015	32 bits Float	100 MHz	Virtex7 VX485T	1.33 GFLOP	75,900	2800	61.62 GFLOPS	61.62	8.12E-04
2016	32 bits Float	100 MHz	Virtex7 VX485T	1.33 GFLOP	75,900	2800	84.2 GFLOPS	84.2	11.09E-04
Proposed Solution	48 bits Fixed	115 MHz	Zynq-7000 AP SoC	0.53 GMACCs	53,200	220	23 GMACCs	46	8.70E-04

Speedup Comparison:

Proposed accelerator design runs at 120 MHz on PL achieving a peak performance of approximately 46 GOPs. Details of resources utilized is given below in Table 6.1. For a tiled Convolution proposed accelerator design achieves a speed up of 10 times compared to base software implementation on a Quad Core ARM Cortex-A72 @ 1.5GHz CPU.

CHAPTER 7: CONCLUSION

This study discusses design and implementation of an FPGA-based Accelerator Design for Deep Convolutional Neural Network. An efficient Accelerator design of CNN on FPGA is highly dependent on the architecture of network to be implemented. VGG-16 style networks are most suitable networks to be implemented on FPGA. They have uniform architectures. Due to limitation of resources on Zynq Zedboard state of art VGG-16 network could not be mapped.

CNNs design and implementation was carried out from scratch in this study. It presents a fully working architectural template on the Xilinx Zynq Zedboard platform. This Study also presents the feasibility implementing CNN models on FPGA in detail. Result of proposed design methodology have shown significant performance gain. But this work is far from finished article but it does provide a potential for number of different opportunities for achieving further breakthroughs in terms of throughput and power efficiency measures. Zynq Zedboard doesn’t have huge amount of computational resources required for further work on CNNs. But this work can still serve as a guidance for further design space exploration of CNNs on FPGA as a platform.

References:

[1] O. M. Parkhi, K. Simonyan, A. Vedaldi, and A. Zisserman. A compact and discriminative face track descriptor. In Proc. CVPR, 2014.
[2] S. Herculano-Houzel, “The human brain in numbers: A linearly scaled-up primate brain”, Frontiers in Human Neuroscience, vol. 3, no. 31, 2009 (cit. on p. 3)
[3] A. Karpathy. (2016). Stanford University CS231n: Convolutional Neural Networks for Visual Recognition, [Online]. Available: http://cs231n.github.io (visited on April. 30, 2020) (cit. on pp. 3–7).
[4] D. Schwind, C. Mayer, and S. Willi, „Origami: Design and implementation of a convolutional neural network accelerator ASIC “, Semester Thesis, ETH Zürich, 2015 (cit. on pp. 4, 6, 17).
[5] I. Goodfellow, Y. Bengio, and A. Courville, „Deep learning “, Book in preparation for MIT Press, 2016 (cit. on pp. 3, 5).
[6] Y. LeCun, Y. Bengio, and G. Hinton, „Deep learning “, Nature, vol. 521, no. 7553, pp. 436–444, 2015 (cit. on p. 5).
[7] Lecun, Y.; Bottou, L.; Bengio, Y.; Haffner, P., "Gradient-based learning applied to document recognition," Proceedings of the IEEE, vol.86, no.11, pp.2278-2324, Nov 1998.
[8] LeCun, Y.; Bottou, L.; Orr, G.; Muller, K., “Efficient BackProp”, in Orr, G. and Muller K. (Eds), Neural Networks: Tricks of the Trade, Springer, 1998.
[9] C. Farabet, B. Martini, P. Akselrod, et al., „Hardware accelerated convolutional neural networks for synthetic vision systems “, in Proceedings of 2010 IEEE International Symposium on Circuits and Systems, IEEE, 2010, pp. 257–260 (cit. on p. 6).
[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, „ImageNet classification with deep convolutional neural networks “, in Advances in Neural Information Processing Systems 25, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2012, pp. 1097–1105
[11] S. Ioffe and C. Szegedy, „Batch normalization: Accelerating deep network training by reducing internal covariate shift “, CoRR, vol. abs/1502.03167, 2015 (cit. on pp. 7, 9).
[12] R. Al-Rfou, G. Alain, A. Almahairi, et al., „Theano: A python framework for fast computation of mathematical expressions “, CoRR, vol. abs/1605.02688, 2016 (cit. on p.8).
[13] E. Battenberg, S. Dieleman, D. Nouri, et al., Lasagne, https://github.com/Lasagne/Lasagne, 2014 (cit. on p. 8).
[14] F. Chollet, Keras, https://github.com/fchollet/keras, 2015 (cit. on p. 8).
[15] R. Collobert, K. Kavukcuoglu, and C. Farabet, „Torch7: A Matlab-like environment for machine learning “, in BigLearn, NIPS Workshop, 2011 (cit. on p. 8).
[16] Martin Abadi, Ashish Agarwal, Paul Barham, et al., TensorFlow: Large-scale machine learning on heterogeneous systems, Software available from tensorflow.org, 2015 (cit. on p. 8).
[17] Y. Jia, E. Shelhamer, J. Donahue, et al., „Caffe: Convolutional architecture for fast feature embedding “, ArXiv preprint arXiv:1408.5093, 2014 (cit. on p. 8).

[18] NVIDIA Corporation, NVIDIA Deep Learning GPU Training System (DIGITS), https://developer.nvidia.com/digits, 2016 (cit. on pp. 8, 25, 80).
[19] The MathWorks, Inc. (2016). Neural Network Toolbox - MATLAB, [Online]. Available: http://mathworks.com/products/neural-network (visited on April. 30, 2020) (cit. on p. 8).
[20] O. Russakovsky, J. Deng, H. Su, et al., „ImageNet Large Scale Visual Recognition Challenge ", International Journal of Computer Vision (IJCV), vol. 115, no. 3, pp. 211–252, 2015 (cit. on p. 9).
[21] A. Karpathy. (2014). What I learned from competing against a ConvNet on ImageNet, [Online]. Available:http://karpathy.github.io/2014/09/02/what-i-learned-from-competingagainst-a-convnet-on-imagenet/ (visited on Jul. 18, 2016) (cit. on pp. 1, 9).
[22] H. Kaeslin, Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication, 1st ed. Cambridge University Press, 2008 (cit. on pp. 13, 14).
[23] Xilinx Inc. (2016). Xilinx: What is an FPGA? Field Programmable Gate Array, [Online]. Available: http://www.xilinx.com/training/fpga/fpga- field- programmable-gate- array.htm (visited on Jul. 19, 2016) (cit. on p. 13).
[24] Xilinx UG998: Introduction to FPGA Design with Vivado High-Level Synthesis, [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/
[25] M. Fingeroff, High-Level Synthesis Blue Book. Xlibris Corporation, 2010 (cit. on pp. 14, 40, 44).
[26] NVIDIA Corporation. (2016). NVIDIA GeForce GTX Titan X Specifications, [Online]. Available:
http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x/specifications
(visited on May, 2020) (cit. on p. 15)
[27] NVIDIA Tegra X1 Whitepaper , (2016), [Online]. Available: http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf
(visited on Jul. 20,2016) (cit. on p. 15).
[28] NVIDIA Launches Tegra X1 Mobile Super Chip. Press release, [Online]. Available: http://nvidianews.nvidia.com/news/nvidia-launches-tegra-x1-mobile-super-chip (visited on May, 2020) (cit. on p. 16).
[29] M. Parker, Altera Corporation. (2014). Understanding Peak Floating-Point Performance Claims, [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01222-understanding-peak-floating-point-performance-claims.pdf (visited on May, 2020) (cit. on pp. 16, 23).
[30] Xilinx Inc. (2016). DSP Solution | Maximum DSP Capabilities, [Online]. Available: http://www.xilinx.com/products/technology/dsp.html (visited on May, 2020) (cit. on pp. 16,23, 33).
[31] Performance and resource utilization for floating-point v7.1, [Online]. Available: http:/ /www.xilinx.com/support/documentation /ip_documentation/ru/ floating-point.html (visited on May, 2020) (cit. on pp. 16, 32).
[32] Xilinx Power Estimator (XPE) 2016.2 Virtex UltraScale+, [Online]. Available: http://www.xilinx.com/publications/technology/powertools/UltraScale_Plus_XPE_2016_2.xlsm (visited on May, 2020) (cit. on p. 16).

[33] Cl´ement Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun. Cnp: An fpga-based processor for convolutional networks. In 2009 International Conference on Field Programmable Logic and Applications, pages 32–37. IEEE, 2009.
[34] Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, Srimat Chakradhar, Igor Durdanovic, Eric Cosatto, and Hans Peter Graf. A massively parallel coprocessor for convolutional neural networks. In 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, pages 53–60. IEEE, 2009.
[35] Srihari Cadambi, Abhinandan Majumdar, Michela Becchi, Srimat Chakradhar, and Hans Peter Graf. A programmable parallel accelerator for learning and classification. In 2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), pages 273–283. IEEE, 2010.
[36] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In Proceedings of the 37th annual international symposium on Computer architecture, pages 247–257, 2010.
[37] Maurice Peemen, Arnaud AA Setio, Bart Mesman, and Henk Corporaal. Memory-centric accelerator design for convolutional neural networks. In 2013 IEEE 31st International Conference on Computer Design (ICCD), pages 13–19. IEEE, 2013.
[38] Jocelyn Cloutier, Eric Cosatto, Steven Pigeon, Francois R Boyer, and Patrice Y Simard. Vip: An fpga-based processor for image processing and neural networks. In Proceedings of Fifth International Conference on Microelectronics for Neural Networks, pages 330–336. IEEE, 1996.
[39] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pages 161–170, 2015.
[40] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. Design space exploration of fpga-based deep convolutional neural networks. In 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC), pages 575–580. IEEE, 2016.
[41] G. Lacey, G. W. Taylor, and S. Areibi, „Deep learning on FPGAs: Past, present, and future “, ArXiv preprint arXiv:1602.04283, 2016 (cit. on p. 16).
[42] A. Putnam, A. Caulfield, E. Chung, et al., A reconfigurable fabric for accelerating large-scale datacenter services, Jun. 2014 (cit. on p. 17).
[43] K. Ovtcharov, O. Ruwase, J.-Y. Kim, et al., Accelerating deep convolutional neural networks using specialized hardware, Feb. 2015 (cit. on pp. 17, 18).
[44] Altera Corporation. (2014). Altera and Baidu collaborate on FPGA-based acceleration for cloud data centers. Press release, [Online]. Available: http://newsroom.altera.com/press-releases/altera-baidu-fpga-cloud-data-centers.html (visited on Jul. 20, 2016) (cit. on p. 17).
[45] S. Higginbotham, The Next Platform. (2016). Google takes unconventional route with homegrown machine learning chips, [Online]. Available: http://www.nextplatform.com/2016/05/19/google-takes-unconventional-route homegrown-machinelearning-chips (visited on May, 2020) (cit. on p. 17).

[46] Y. Chen, T. Luo, S. Liu, *et al.*, „DaDianNao: A machine-learning supercomputer “, in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2014, pp. 609–622 (cit. on p. 17).
[47] L. Cavigelli, D. Gschwend, C. Mayer, *et al.*, „Origami: A convolutional network accelerator “, in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, ser. GLSVLSI ’15, Pittsburgh, Pennsylvania, USA: ACM, 2015, pp. 199–204 (cit. on p. 17).
[48] Integrated Systems Laboratory, D-ITET, ETH Zürich. (2016). IIS Projects Proposal: FPGA System Design for Computer Vision with Convolutional Neural Networks, [Online]. Available: http://iisprojects.ee.ethz.ch/index.php/FPGA_System_Design_for_Computer_Vision_with_Convolutional_Neural_Networks (visited on May, 2020) (cit. on p. 17).
[49] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, „Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks “, in *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, IEEE Computer Society, 2016, pp. 262–263 (cit. on p. 18).
[50] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network (2013). arXiv preprint arXiv:1312.4400, 396, 2013.
[51] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.
[52] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 1– 9, 2015.
[53] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.
[54] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. arXiv preprint arXiv:1602.07360, 2016.
[55] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. Commun. ACM, 52(4):65{76, Apr. 2009.
[56] Xilinx UG902: Vivado Design Suite User Guide, High-Level Synthesis, [Online]. Available: http://www.xilinx.com/support/documentation/ sw_manuals/xilinx2016_2/ug902-vivado-high-level-synthesis.pdf (visited on May, 2020) (cit. on pp. 14, 43, 44,52).
[57] M. Fingeroff, High-Level Synthesis Blue Book. Xlibris Corporation, 2010 (cit. on pp. 14, 40, 44).
[58] Alexander Mamaev. Flower recognition dataset-kaggle. https://www. kaggle.com/alxmamaev/flowers-recognition/metadata, (visited on May 2020).
[59] Siddharth Das. CNN Architectures: LeNet, AlexNet, VGG, GoogLeNet, ResNet and more. https://medium.com/analytics-vidhya/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5

APPENDIX A

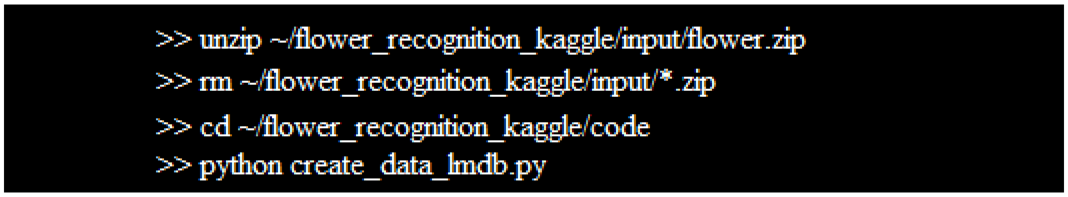
Ubuntu Terminal Commands to setup Caffe

```
Command 1: sudo apt-get upgrade
Command 2: sudo apt-get update
Command 3: sudo apt-get install build-essential git cmake pkg-config libjpeg8-dev libjasper-dev  \
           libpng12-dev libgtk2.0-dev libavcodev-dev libavformat-dev libswscale-dev libv41-dev  \
           libtiff-dev gfortran
Command 4   : sudo apt-get install libatlas-base-dev
Command 5   : wget https://bootstrap.pypa.io/get-pip-py
Command 6   : sudo python get-pip.py
Command 7   : sudo apt-get install python2.7-dev
Command 8   : pip install numpy
Command 9   : cd opencv
Command 10 : mkdir build
Command 11 : cd build
Command 12 : cmake  -D CMAKE_BUILD_TYPE=Release  \
                     -D CMAKE_INSTALL_PREFIX = /usr/local/  \
                    -D CMAKE_INSTALL_C_EXAMPLE = ON \
                    -D CMAKE_INSTALL_PYTHON_EXAMPLE = ON \
                    -D OPENCV_EXTRA_MODULES_PATH = ~/opencv-contrib/modules / \
                    -D BUILD_EXAMPLE=ON
Command 13 : make
Command 14 : make j16
Command 15 : sudo make install
Command 16 : sudo ldconfig
Command 17:  python
Command 18:  import cv2
Command 19: cv2.__version__
           : '3.0.0'
Command 20 :exit
Command 21: sudo apt-get install libprotobuf-dev libleveldeb-dev libsnappy-dev libopencv-dev  \
            libboost-all-dev libhdf5-dev libflags-dev libgoogle-glog-dev liblmdb-dev protobuf-compiler
Command 22:  git clone https://github.com/BVLC/caffe
Command 23:  cd caffe
Command 24:  cp makefile.conifig.example makefile.config
Command 25:  sudo vim makefile.config
Command 26:  USE_CUDNN:=1
Command 27:  OPENCV_VERSION:=3
Command 28:  PYTHON_INCLUDE:= /usr/include/python2.7 \
             /usr/local/lib/python-2.7/dist-packages/numpy/core/include \
Command 29: WITH_PYTHON_LAYER: = 1
Command 30: INCLUDE_DIR: = $(PYTHON_INCLUDE) /usr/local/include /usr/local/hdf5/serial
Command 31: LIB_DIR : = $(PYTHON_LIB) / usr/local/lib
Command 32:  make all & make test & make runtest & make pycaffe
Command 33: python
Command 34: import caffe
Command 35: caffe.__version__
          :  '1.0.0'
```

APPENDIX B

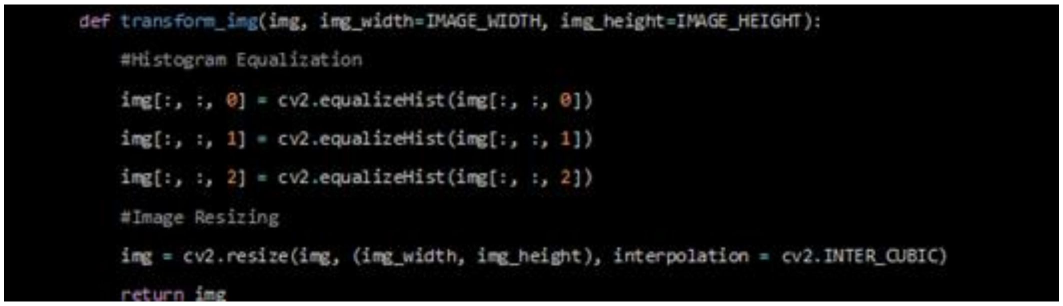
Data Pre-Processing Codes

Step 1:



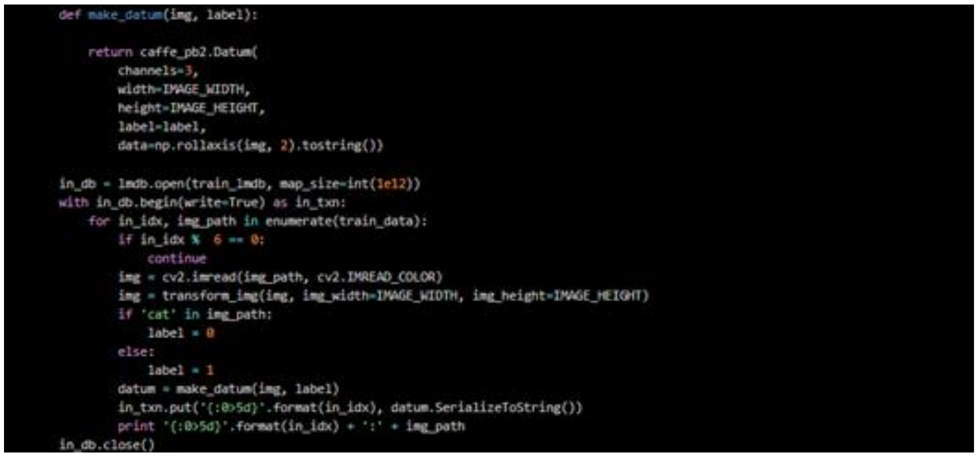
```
>> unzip ~/flower_recognition_kaggle/input/flower.zip
>> rm ~/flower_recognition_kaggle/input/*.zip
>> cd ~/flower_recognition_kaggle/code
>> python create_data_lmdb.py
```

Step 2:



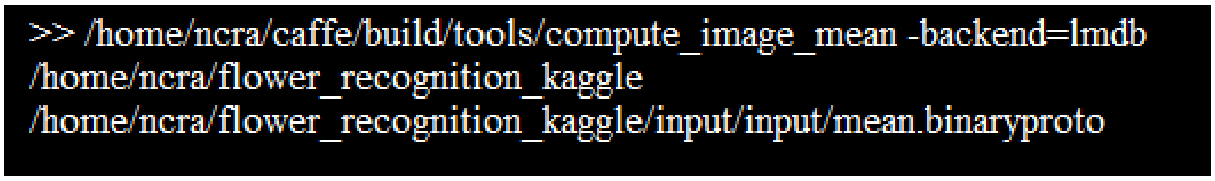
```
def transform_img(img, img_width=IMAGE_WIDTH, img_height=IMAGE_HEIGHT):
    #Histogram Equalization
    img[:, :, 0] = cv2.equalizeHist(img[:, :, 0])
    img[:, :, 1] = cv2.equalizeHist(img[:, :, 1])
    img[:, :, 2] = cv2.equalizeHist(img[:, :, 2])
    #Image Resizing
    img = cv2.resize(img, (img_width, img_height), interpolation = cv2.INTER_CUBIC)
    return img
```

Step 3:



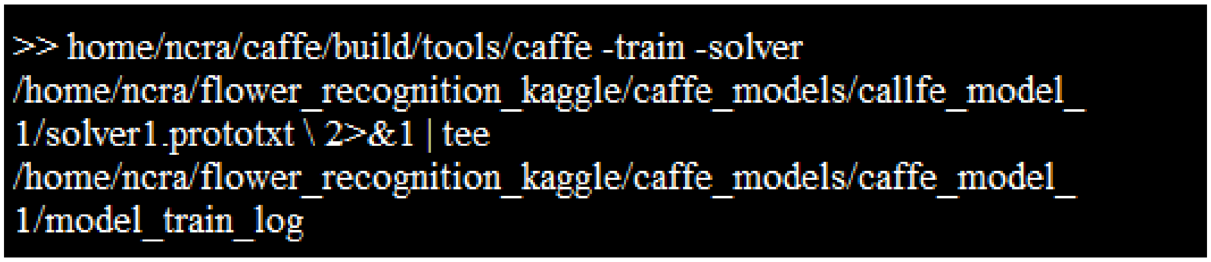
```
def make_datum(img, label):

    return caffe_pb2.Datum(
        channels=3,
        width=IMAGE_WIDTH,
        height=IMAGE_HEIGHT,
        label=label,
        data=np.rollaxis(img, 2).tostring())

in_db = lmdb.open(train_lmdb, map_size=int(1e12))
with in_db.begin(write=True) as in_txn:
    for in_idx, img_path in enumerate(train_data):
        if in_idx %  6 == 0:
            continue
        img = cv2.imread(img_path, cv2.IMREAD_COLOR)
        img = transform_img(img, img_width=IMAGE_WIDTH, img_height=IMAGE_HEIGHT)
        if 'cat' in img_path:
            label = 0
        else:
            label = 1
        datum = make_datum(img, label)
        in_txn.put('{:0>5d}'.format(in_idx), datum.SerializeToString())
        print '{:0>5d}'.format(in_idx) + ':' + img_path
in_db.close()
```

Step 4:



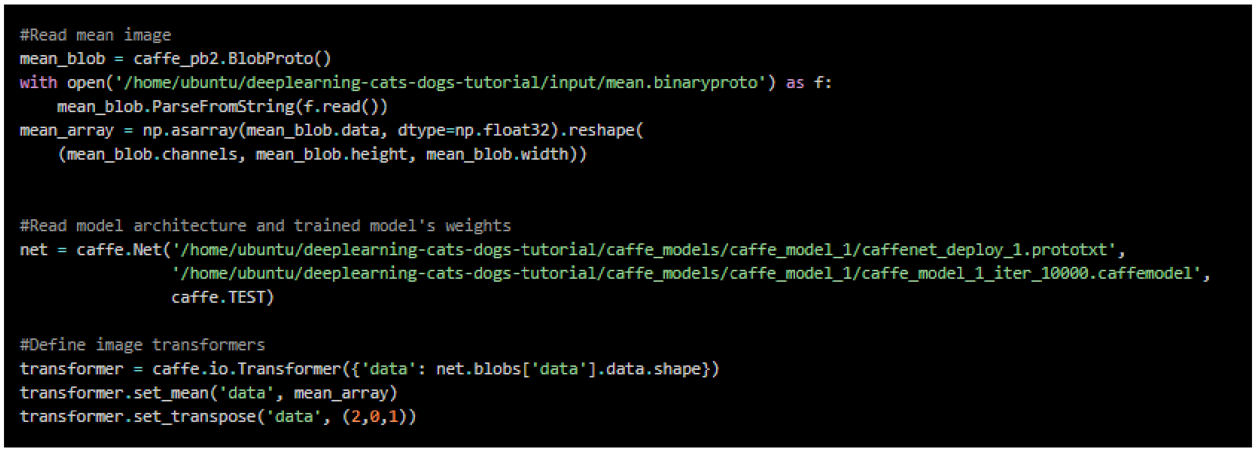
```
>> /home/ncra/caffe/build/tools/compute_image_mean -backend=lmdb
/home/ncra/flower_recognition_kaggle
/home/ncra/flower_recognition_kaggle/input/input/mean.binaryproto
```

Step 5:



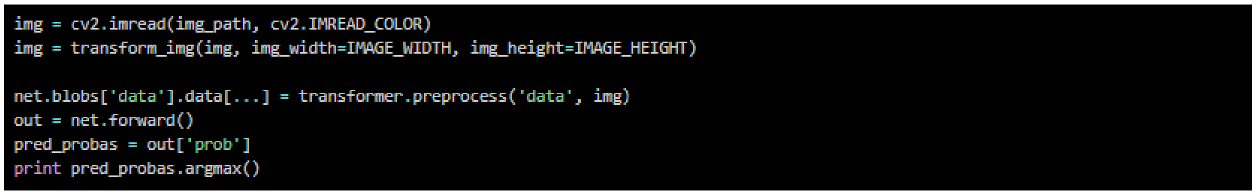
```
>> home/ncra/caffe/build/tools/caffe -train -solver
/home/ncra/flower_recognition_kaggle/caffe_models/callfe_model_
1/solver1.prototxt \ 2>&1 | tee
/home/ncra/flower_recognition_kaggle/caffe_models/caffe_model_
1/model_train_log
```

Step 6:



```
#Read mean image
mean_blob = caffe_pb2.BlobProto()
with open('/home/ubuntu/deeplearning-cats-dogs-tutorial/input/mean.binaryproto') as f:
    mean_blob.ParseFromString(f.read())
mean_array = np.asarray(mean_blob.data, dtype=np.float32).reshape(
    (mean_blob.channels, mean_blob.height, mean_blob.width))


#Read model architecture and trained model's weights
net = caffe.Net('/home/ubuntu/deeplearning-cats-dogs-tutorial/caffe_models/caffe_model_1/caffenet_deploy_1.prototxt',
                '/home/ubuntu/deeplearning-cats-dogs-tutorial/caffe_models/caffe_model_1/caffe_model_1_iter_10000.caffemodel',
                caffe.TEST)

#Define image transformers
transformer = caffe.io.Transformer({'data': net.blobs['data'].data.shape})
transformer.set_mean('data', mean_array)
transformer.set_transpose('data', (2,0,1))
```

Step 7:



```
img = cv2.imread(img_path, cv2.IMREAD_COLOR)
img = transform_img(img, img_width=IMAGE_WIDTH, img_height=IMAGE_HEIGHT)

net.blobs['data'].data[...] = transformer.preprocess('data', img)
out = net.forward()
pred_probas = out['prob']
print pred_probas.argmax()
```