IN-SITU IMPLEMENTATION AND TRAINING OF CONVOLUTIONAL NEURAL NETWORK ON FPGAS

by

Akshay Raju Krishnani

APPROVED BY SUPERVISORY COMMITTEE:

_____

Dr. Benjamin Carrion Schaefer, Chair

_____

Dr. Dinesh Bhatia

_____

Dr. Lakshman Tamil

I would like to thank my father for showing his trust in me and always being a role model to me, my mother for making me a good human being by her constant love and care, and my brother who has always been a good friend of mine. At last, I would like to thank my girlfriend for her care and support.

IN-SITU IMPLEMENTATION AND TRAINING OF CONVOLUTIONAL NEURAL NETWORK ON FPGAS

by

AKSHAY RAJU KRISHNANI, B. TECH.

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

MASTERS OF SCIENCE IN

ELECTRICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT DALLAS

August 2020

ACKNOWLEDGMENTS

I would like to warmly thank my thesis advisor, Dr. Benjamin Carrion Schaefer for his support and guidance. During my first semester at UTD in Fall 2018, I took a course under Dr. Schaefer and was highly impressed by his research work and way of teaching. This highly motivated me to pursue a master's thesis under him. More than a thesis advisor, he has been a mentor for me throughout my master's degree. His positive attitude and optimistic targets embed a passion inside you to learn new things and never give up on anything. He brings out the best in yourself.

I would like to thank Dr. Dinesh Bhatia and Dr. Lakshman Tamil for being a part of my committee and spending their quality time to review my work.

I am also thankful to Jianqi for guiding me during the implementation of the neural network on FPGA, and Farah for helping me understanding the CyberWorkBench software and its various features. Also, I would like to thank members of DARClab - Zhiqi, Zi, and Prattay for their constant help and support.

I am also thankful to my parents and my brother Rahul for showing their confidence in me. I am also thankful to the facility provided by the DARC lab. I am also grateful to the Department of Electrical and Computer Engineering at The University of Texas at Dallas for the research facilities to help me throughout my tenure.

July 2020

IN-SITU IMPLEMENTATION AND TRAINING OF CONVOLUTIONAL

NEURAL NETWORK ON FPGAS

Akshay Raju Krishnani, MS
The University of Texas at Dallas, 2020

Supervising Professor: Dr. Benjamin Carrion Schaefer

The main objective of this thesis is to investigate the efficiency of in-situ trainable Convolutional Neural Networks (CNNs) on modern programmable System-on-Chip (SoC) Field Programmable Gate Arrays (FPGAs) composed of embedded processors and reconfigurable fabric and to study the robustness of the system when faults happen. One particular characteristic of this work is that CNN is developed exclusively using High-Level Synthesis (HLS), particularly in SystemC, generating Verilog code. In this thesis, the feature maps are also being trained on the FPGA, which is traditionally done offline. The CNN architecture is instantiated on the FPGA and weights are trained through the software model on the ARM processor embedded into the FPGA and updated in the architecture through the AXI bus interface.

Moreover, since CNN is implemented in hardware the resource used need to be minimized. This allows to choose a smaller, and cheaper FPGA, as well as reducing the total power consumption. To address this, the effect of bitwidth reduction of the CNN is investigated with respect to the accuracy of handwritten characters recognitions. Finally, the robustness of the CNN is analyzed by breaking internal connection of different neurons studying how the accuracy drops when the fault happens at different layers If the accuracy is reduced, then the CNN is re-trained in-situ to increase the accuracy of the CNN.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF NOMENCLATURES

ALM	Adaptive Logic Module
ARM	Advanced RISC Machine
ASIC	Application-Specific Integrated Circuits
ANN	Artificial Neural Network
BDL	Behavioral Descriptive Language
CPU	Central Processing Unit
CNN	Convolutional Neural Network
CWB	CyberWorkBench
DAC	Digital to Analog Converter
FFN	Feed Forward Network
FPGA	Field Programmable Gate Array
FC	Fully Connected
GPU	Graphical Processing Units
HLS	High-Level Synthesis
MNIST	Modified National Institute of Standards and Technology database
MLP	Multi-Layer Perceptron
RNN	Recurrent Neural Network
RTL	Register Transfer Level
SoC	System on Chip

CHAPTER 1

INTRODUCTION

1.1 Thesis Motivation

Artificial intelligence and deep learning have become one of the key computing paradigms for many applications such as image and speech recognition. These applications are often found in self-driving cars, and game-playing [1]. One of the deep learning algorithms vastly used in pattern recognition is convolutional neural network (CNN) [2]. Deep convolutional neural network algorithms have been successfully adopted in many domains, e.g. computer vision community [3].

These algorithms require a large amount of memory and throughput to perform computationally intensive computations. As CNNs requires convoluting feature maps (filters) over large size images to perform classification, an extremely large number of floating-point operations and model parameters are required. This makes the implementation of the CNN on the Central Processing Unit (CPU), very inefficient in terms of performance while consuming a significant amount of power [4]. Because of the large amount of parallelism, other architectures that can exploit this massive parallelism have been shown to clearly outperform CPUs.

In particular, Graphical Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and Application Specific Integrated Circuits (ASICs) are used to increase the throughput of the design. Among these accelerator platforms, GPUs are widely used due to the availability of various programming frameworks like Caffe and Tensorflow, that simplify the implementation process. Additionally, GPU offers a larger number of floating operations per second compared to CPU [5]. The main downside of implementing CNNs on GPUs is their high energy utilization. To address this issue, FPGAs have been proposed. They can fully customize the architecture reducing the size of the multipliers required and using fixed-point data types instead of the power-hungry floating-point counterparts. Thus, much work has been done in implementing CNNs on FPGAs [4].

One of the main problems with FPGAs is how they are programmed. Typically using low-level hardware description languages such as Verilog or VHDL. To facilitate the design process, High-Level Synthesis (HLS) has been proposed to generate the HDL automatically. HLS takes as input

high-level languages such as C or C++and generates optimized Verilog or VHDL than can in turn be used to program the FPGA. Designing algorithms using high-level language removes the tediousness faced while working with HDL and optimizing the HDL designs [6]. There are currently numerous HLS tools that can efficiently make use of heterogeneous resources in modern FPGAs. Thus, these HLS has recently received much attention in the context of FPGA design [7].

The presented work explores CNN implementation on modern programmable SoCs using HLS including the inference and the training part in situ. The training part is traditionally done offline. In this work, we proposed a self-contained system that can perform the computation and the training on the same chip. This thesis also studies the resource utilization vs. the throughput of resultant CNN. Hardware implementation work on approximate computing across high-level synthesis which might affect the accuracy of the classifiers but at the same time minimizing the area occupied in the final hardware and making it more efficient is also studied. Finally, the robustness of CNN architecture is analyzed by breaking neurons and their connections at different stages and observing its effect on the accuracy of the model.

1.2 Thesis Contribution

In summary, the contributions of this thesis towards implementing CNN on an FPGA can be summarized as follows:

- Implementation and in-situ training of a CNN on the programmable SoC composed of a dual-core ARM processor and programmable logic.
- Optimization framework to reduce the area of the CNN by reducing the bitwidth of the neurons and studying the robustness of the network as compared to the original floating-point version.
- Study the robustness of the Convolutional Neural Network by investigating when different internal connections with the CNN break.

The application targeted in this work is the handwriting character identification using the MNIST dataset. Since the required architecture is comparatively narrow and easy to train, it gives a better opportunity to analyze the loss in classification accuracy seen when one of the neurons contributing directly or indirectly towards classifier is made to stuck-at-0.

CHAPTER 2

NEURAL NETWORKS

2.1 Introduction

Neural networks mimic how biological neural connection process information. In biological connections, neurons are the basic unit of computation and receive input from other nodes computing the output. In case that the output is above a certain level, the neuron fires. Similarly, in neural networks, each input is associated with weights, these weights signify the relative importance of other inputs. The ultimate decision on the firing of the neuron is decided by the activation function of each neuron [8].

As shown in Figure 2.1, a basic neural network consists of three layers [2]. These layers are:

- Input layer – Form of multidimensional input vector obtained from an image (in our case it is an image but it could be any information that needs to be processed)
- Hidden layer – Performs decision making tasks and accordingly weighs changes to improve the learning ability of the model
- Output Layer – Based on data obtained, classifies the output into different classes

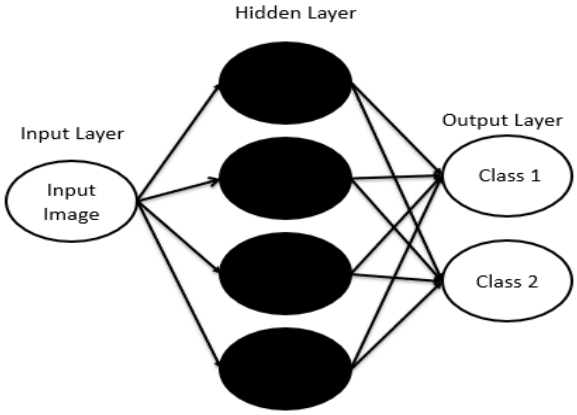

Figure 2.1: Basic Structure of Neural Network

2.2 Types of Neural Networks

As the application varies, so does the architecture of the neural network. Here, three types of neural networks are discussed which find their usage in many applications [9]. Many neural networks are either derived or are modified versions of these three networks.

Types of neural networks are:

1. Feedforward Neural Network
2. Recurrent Neural Network
3. Convolutional Neural Network

The next subsections briefly describe these networks.

2.2.1 Feedforward Neural Network (FNN)

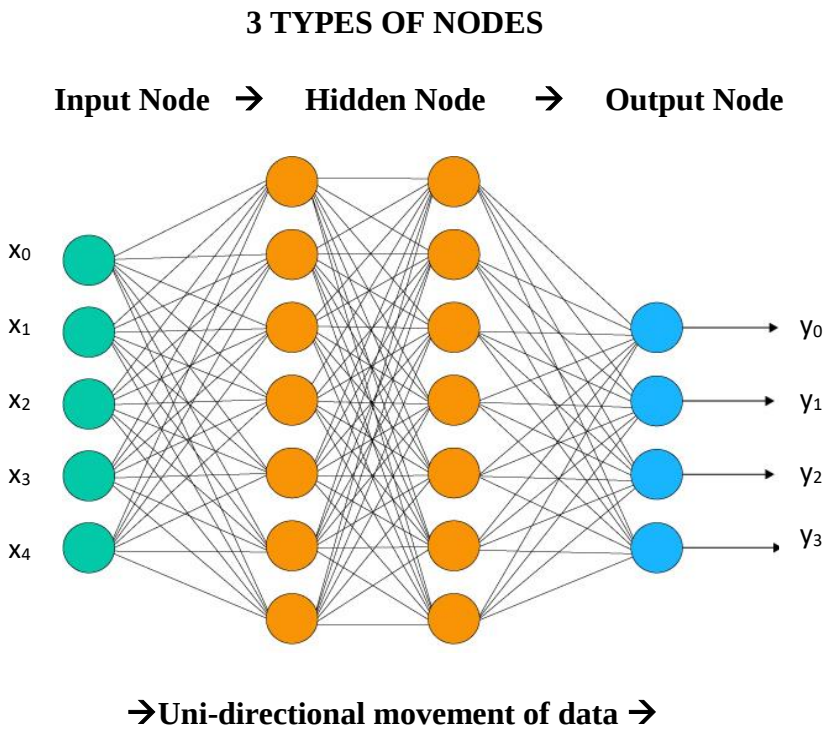

Figure 2.2: Structure of Feedforward Neural Network [10]

FFN is the basic component of the neural network operating on supervised learning. These are often termed as Multi-level Perceptron (MLP) and consist of a uni-directional data flow. Figure 2.2 shows the basic components of the network, which are (1) the input layer, (2) hidden layer, and (3) the output layer. These layers are all interconnected to all of its neurons. The hidden layers help in introducing non-linearity in the data. In Mathematical sense, for a classifier, where y = f(x) mapping an input x to output y, the network will define a mapping as y = f(x,w) where w would be learnable features resulting in the proper identification of the pattern. The learnable features are termed as weights and biases [11]. These learnable features help in adjusting the connection with the help of a learning algorithm. After mapping the data with learnable features, it is passed to the activation functions. This helps in activating certain features and develop the ability to efficiently map data onto certain dimensions. These functions introduce non-linearity in the data. The most widely used activation functions are ReLU, tanh, sigmoid function, and softmax function.

2.2.2 Recurrent Neural Network (RNNs)

RNNs are a subset of feedforward neural networks, further improved by the addition of recurrent edges which help in taking into account the previous observations. They form cycles including the self-connections. Just like a basic neural network, RNN also consists of a hidden layer, and after the hidden layer, the data is passed through activation function and further through the output layer for classification. As shown in Figure 2.3, the architecture takes into account the data at time step t-1 as well as input passed at time t. This helps the network to include previous data as well as current data [12].

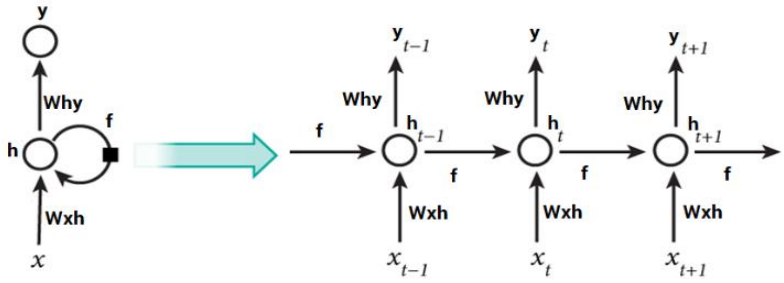

Figure 2.3: **Left:** RNN architecture flow, **Right:** Unfolded notation for RNN[12]

2.2.3 Convolutional Neural Networks (CNNs)

The most popular deep learning topologies for many applications and in particular applications that require image processing are CNNs and FNNs. However, in many complex applications, CNNs are preferred over FNNs. The reason behind the preference of CNNs over FNNs is that in FNNs there is a learnable parameter for each input or pixel of the image. This increases the number of parameters for large images making it unmanageable. As the size of input increases, the number of fully connected layer increases which increases the number of weights to be trained which might further result in redundancy and inefficiency. To overcome these issues, CNNs were introduced where filter parameters are used to find patterns that can be applied over any size of an image. It leverages the fact that nearby pixels are correlated more compared to distant pixels and can be used to find a pattern. Also, these filters are shared across the data which makes it easy to find the pattern across the image with a limited number of parameters [2].

CNNs are inspired by the visual perception of living organisms. During the 1990s, LeCun et al. [13] published a paper proposing the modern framework for CNNs, which was later updated by them in [14]. Here multi-layer neural networks were developed to classify handwritten digits called LeNet-5. It was also observed that an increasing number of layers and using the non-linear function can result in better approximation and better feature extraction [15].

In this thesis, we use a CNN architecture to classify the images into 10 classes, each class consisting of digits from 0-9. This method consists of training the features maps (weights) from a large dataset. Here feature maps are the filters (of matrix X*Y) that cover the area of images and extract the pattern from it. Based on the expected output, the filters are being trained and updated to recognize the pattern.

The basic architecture of CNN consists of convolutional layer, pooling layer, and fully-connected layer (often called hidden layer).

Convolutional layer:

This first layer is responsible for approximately 90% of the computation involved in Covnet [44]. The convolutional layer is the stage where feature maps come into the picture, often termed as learnable kernels. These kernels slide over the input data and are multiplied with the pixel values of the image [2]. This operation initiates from the top left corner of the 2D array and parses through the width till the last element with a certain stride value. Then it moves down to the beginning of the next row, here the value by which it shifts to the next row should be equal to the stride value used while covering the previous width. Again, the parsing is done along the current width, and the same process is repeated until the entire image is navigated.

The convolution function in Covnet is:

$$Conv(a,b) = (\sum_{u=0}^{2} \sum_{v=0}^{2} input\_image[a+u][b+v].filter[u][v]) + bias$$

In the above function the size of the input image is (a+2)*(b+2). The input image is convoluted through the filter and biases are added, resulting in the convolution layer. In case that there are more features to be extracted from the image data, then the number of filters are increased proportionally. A bias value is also introduced during the convolution stage which allows the shifting of activation function to better fit the data. In absence of any bias, the model will be trained through an origin only (since there will not be any offset) added, which is not the real-world scenario [16].

After the convolution is done, data is passed through an activation function to bring non-linearity in the data.

Pooling Layer:

After the convolutional layer, a pooling is performed on the data. The main advantage of using a pooling layer is a reduction in the size of the input data along with the spatial dimensionality. This further limits the number of parameters and avoid the overfitting of the data. It operates on each feature map independently [2].

The most commonly used pooling layer is with a size of 2*2 applied along with the stride of 2, which results in downsampling every depth slice by 2 along width and heights. This further results in reducing the overall size of 75%. There are 2 types available: Max pooling and average pooling. Among these, max pooling is preferred due to better results observed in the past [17].

1. **Average pooling [17]:**

In average pooling, the average of the elements is taken along which downsampling is performed.

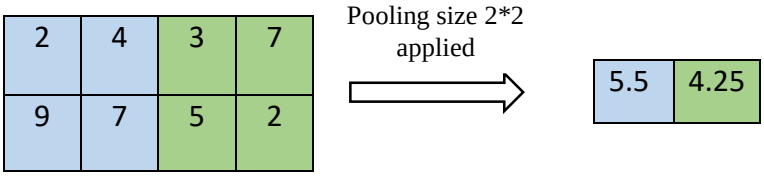

Figure 2.4: Average pooling applied to the Input data

Figure 2.4 displays an example of how average pooling is applied to the data with stride 2.

For output array obtained in the figure 2.4, the elements are obtained as,

Element 1 = avg{2,4,9,7} = 5.5

Element 2 = avg{3,7,5,2} = 4.25

2. **Max pooling [17]:**

In max pooling, the maximum of the elements is taken along which downsampling is performed.

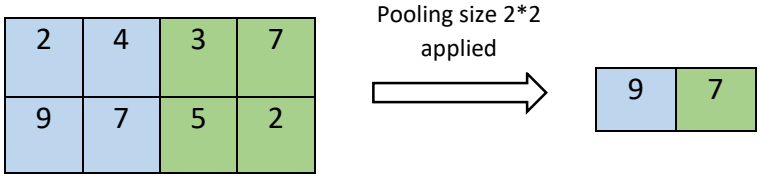

Figure 2.5: Maxpooling applied on the Input data

Figure 2.4 displays an example of how max-pooling is applied to the data with stride 2.

For output array obtained in figure 2.4, the elements are obtained as,

Element 1 = max{2,4,9,7} = 9

Element 2 = max{3,7,5,2} = 7

Fully Connected Layer:

The fully connected (FC) layer is a feed-forward network. Here each unit of the pooling layer is connected to each unit of a fully connected layer. FC is a 1D layer, hence before connecting the pooling layer to the FC layer, the layer must be flattened which means that 2D array is converted into a 1D array. FC layers are commonly used as a classification layer at the end of architecture followed by activation function and the output acts as a class score [2]. A CNN architecture can consist of one or more fully connected layers. Mathematically, the output of fully connected can be defined as:

$$Fout[i] = (\sum_{j=0}^{k} Fin[j] * weights[i][j]) + bias[i]$$

Here Fout is the output layer obtained and Fin is the input to the hidden layer. When FC is used as an output classification layer, softmax classification is used to compare with each class. The output of each neuron is then connected to an activation function that establishes if the neuron will fire or not. There are many different types of activation functions. The main ones are summarized below.

Types of Activation function:

1. Sigmoid Function: It has a range between 0 to 1 and has the form of an S-shaped curve. Due to some drawbacks, the function went out of popularity, such as vanishing of the gradient, slow convergence, and difficult to perform optimization because the output is not zero centered [18]. Figure 2.6 shows the curve for output obtained using sigmoid function.

   Mathematical expression:

$$f(x) = \frac{1}{1 + e^{-x}}$$

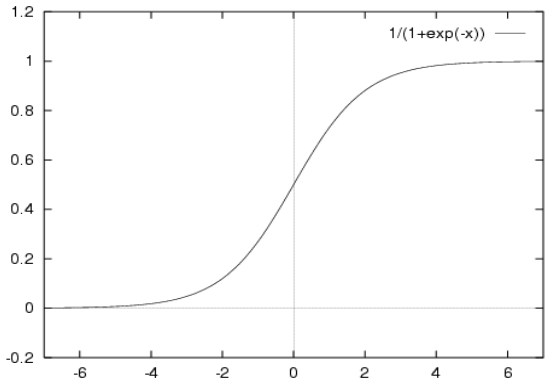

Figure 2.6: Curve for Sigmoid Function

2. Tanh: In the Tanh function, the output is zero centered, hence it overcomes the issue seen while working sigmoid function but fails to resolve the vanishing gradient issue [18]. Figure 2.7 shows the output curve obtained while passing the data through Tanh function.

   Mathematical expression:

$$f(x) = tanh(x)$$

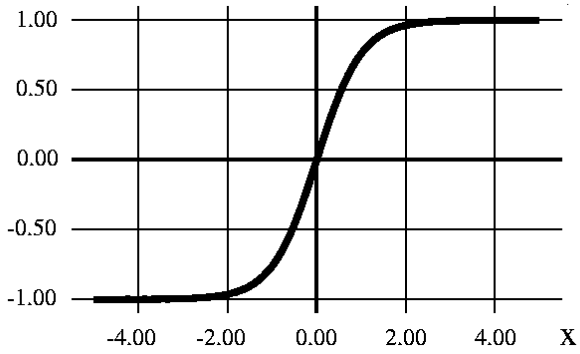

Figure 2.7: Curve for Tanh function [18]

3. ReLU : It is the most preferred function in deep learning and comparing its performance with tanh activation function, leads to convergence improvements of up to by 6 times [18][39]. It further resolves the issue of vanishing gradient [39].

Function for ReLU is R(x) = max(0,x)

Here x is the input data to activation function and according to the function if x is less than 0 then R(x) will be 0 otherwise R(x) = x. Figure 2.8 displays the graph for the same with varying value of x.

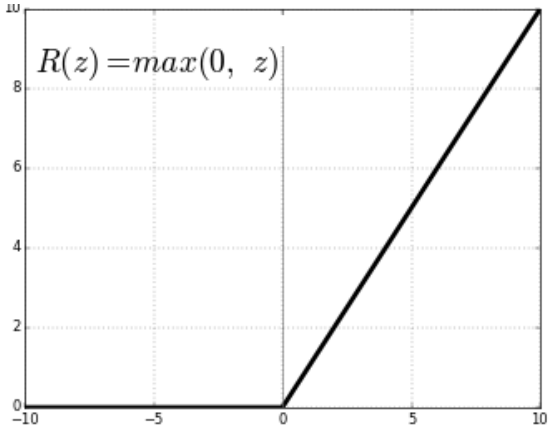

Figure 2.8: Curve for ReLU function [18]

4. Softmax Classification: It is another type of activation function preferred to classify the output. It calculates the probability for each class and produces a range of values between 0 and 1 for each class. The sum of probabilities for each class is equal to 1 [18].

   Mathematical exp:

$$S(y[i]) = \frac{e^{y[i]}}{\sum_j e^{y[i]}}$$

   Here, *y[i]* are the output scores obtained at the last layer of the neural network. S(*y[i]*) is the classification obtained using the output layer scores. The main advantage of using softmax is it can be used for a multiclassification task [18].

Regularization:

Due to a huge number of parameters and a lot of training data available there is a chance overfitting of parameters may take place. Overfitting happens when a model is trained too well. That means when the model learns details from noisy data which causes random fluctuation in the training

data. This negatively affects the accuracy of the model. Hence to resolve the issue of overfitting regularization methods are used. Regularization adds a penalty to the parameters with the increase in the complexity of the model [19]. There are mainly 2 types of regularization methods are used:

1. L1 Regularization:

Also called Lasso Regression, it adds an absolute value of the feature to be trained as a penalty to loss function used [19].

Mathematical Expression:

$$\text{New loss obtained} = \text{loss function} + \lambda * \sum_{j=0}^{n} w[j]$$

λ = Regularization Parameter

w[j] = Weight being trained

2. L2 Regularization:

Also called Tikhonov Regularization, here the square value of the feature to be trained is added as a penalty to the loss function [19].

Mathematical Expression:

$$\text{New loss obtained} = \text{loss function} + \lambda * \sum_{j=0}^{n} w[j]\text{^}2$$

λ = Regularization Parameter

w[j] = Weight being trained

In the proposed model, L1 Regularization is used which helps in avoiding overfitting at a greater extend.

CHAPTER 3

HIGH-LEVEL SYNTHESIS

3.1 Introduction

With the increase in complexity of integrated circuits, it is becoming more and more challenging to create designs using low-level hardware description languages. Hence, due to this increase in the complexity of SoC design, novel approaches are sought to create designs at higher abstraction levels rather than the RT-level. One such approach is HLS. HLS enables the automatic synthesis of untimed higher-level language into low-level RTL design to be used in ASIC or FPGA design. Optimization of the synthesis can be done based on performance, power, and cost. Although past HLS efforts failed to provide a high-quality solution, modern HLS tools have shown to rival hand-optimized RTL designs [20]. Some of the reasons why HLS has become so popular are:

1. HLS simplifies the design complexity and increases productivity
2. It allows to re-use the behavioural description resulting in better productivity
3. It provides better verification capability

Hence, more and more FPGA designers are utilizing HLS tools to synthesize low-level designs. Some of the applications are 3 G/4 G wireless systems [21], [22], aerospace applications [23], image processing [24], lithography simulation [25], and cosmology data analysis [26].

HLS was introduced into the market way before FPGAs were developed, majorly focusing on ASIC designs. Among the initially released HLS tool, researchers from Carnegie Mellon University during the 1970s released a tool named Carnegie-Mellon University design automation (CMU-DA) [27] [28]. Here designs were specified at behaviour level with the help of instruction set processor specification (ISPS) language which is further translated into data-flow representation [29]. In the following years, many HLS tools were released for research and prototyping. These tools divided synthesis into a couple of steps, including code transformation, module selection, operation scheduling, datapath allocation, and controller generation which required further work to resolve issues related to each step. Another reason the synthesis of RTL did not gain much popularity was due to the lack of maturity in the RTL synthesis tool [20].

Recently many HLS tools have been developed at both academic and industrial levels as discussed in table 3.1. Many tools now focus on using C/C++ language to make tools accessible at system level design rather than only HDL based. This enables the reach to develop software as well as hardware using the same model resulting in software/hardware co-design. C-based design help in leveraging software compilers for better parallelization and optimization in synthesis tools.

Table 3.1: Overview of Some HLS Tools Used at Industrial and Academic Level

Tool	Device Supported	Language Supported
Xilinx Vivado HLS	FPGA	C/C++/SystemC
Cadence Stratus HLS	FPGA/ASIC	C/C++/SystemC
Mentor Catapult	FPGA/ASIC	C/C++/SystemC
NEC CyberWorkBench	FPGA/ASIC	C/C++/SystemC

3.2 HLS Design flow

Considering the initial stage of design abstraction to the low-level implementation, the typical HLS process follows these stages[6], also shown in Figure 3.1:

1. Compilation of behavioural description
2. Resource allocation for the design
3. Scheduling of the design along with the clock cycles
4. Binding operations to the functional units and variables to storage elements
5. Generate RTL for design specifications

In the HLS design flow, the design is initiated with the specification of the application to be implemented in the form of hardware. This includes writing a functional description which includes the inputs and outputs. The behavioural description can be done in Behavioural Descriptive Language (BDL) C/C++ or SystemC to achieve the desired operation at higher-level abstraction.

Input and Output data flow can be operated concurrently or sequentially depending on the type of application. These functional blocks create a formal representation of the input specifications with all the code optimizations [6]. Figure 3.1 shows the flow of HLS design, including all the major steps required.

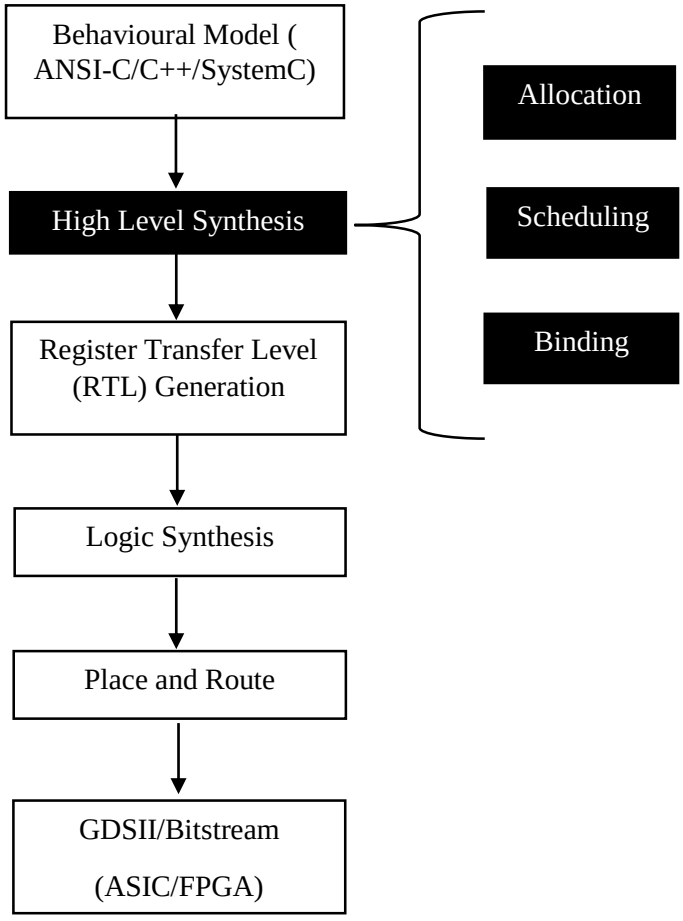

Figure 3.1: HLS Design Flow

Figure 3.2 shows the main steps in HLS. In particular the allocation, scheduling and binding.

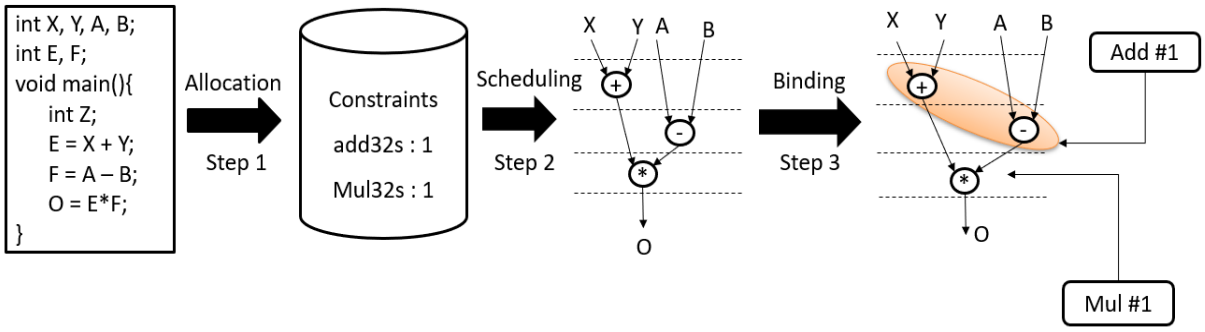

Figure 3.2: Flow of Allocation, Scheduling, and Binding

Compilation and Modelling:

The parsed behavioural description is represented as a the data flow graph (DFG) where each node represents an operation/function and the edges in between the nodes signifies input, output, and temporary variables [30]. Control dependencies often create complexities and can be removed by unrolling loops into non-iterative code blocks and conditional statements resolved into multiplexers. By removing dependencies in DFG, parallelism is introduced in the design implementation. However, such transformation creates a requirement of a huge amount of memory during the process of synthesis. A major drawback of DFG is the lack of support for loops with unbounded iteration counts and non-static control statements [6].

Allocation:

As shown in the Figure 3.2, during the stage of allocation, the tool decides the type and number of resources to be allocated to optimize the design according to the constraints specified. Some of the HLS tools may add a couple of components during the stage of scheduling and binding, such as the addition of connectivity components (such as bus connection or point-to-point connection among components). These components are chosen from the RTL component library and for each specification, at least one component is chosen. At the end of this stage, it must also report the other characteristics like area, delay, and power which would be used in the next stages [6].

Scheduling:

Scheduling of operations plays a major role while synthesizing designs using HLS. It can be defined as a procedure to assign a cycle or control state to each operation as shown in Figure 3.2 . There are various parameters taken into consideration while performing the assignment, such as clock frequency, time taken by the operation to complete, and user-specified directive. Scheduling is done to avoid violation of any prior constraints defined while specifying design details. For example for an operation, a = b op c, where b and c in the form of input which is read from the sources and is fed into a functional unit that further executes required operation op, and the obtained result is stored to a destination (storage or functional units). Depending on the type of operation to be mapped, it is scheduled within one cycle or over several cycles. In the case of multiple operations, they can be scheduled parallelly provided there is no data dependency and availability of sufficient resources [6]. Let's consider an example with the target computation, y = x*a+b+c .

Below is the Example:

```
int example(char x, char a, char b, char c) {
char y;
y = x*a+b+c;
return y;
 }
```

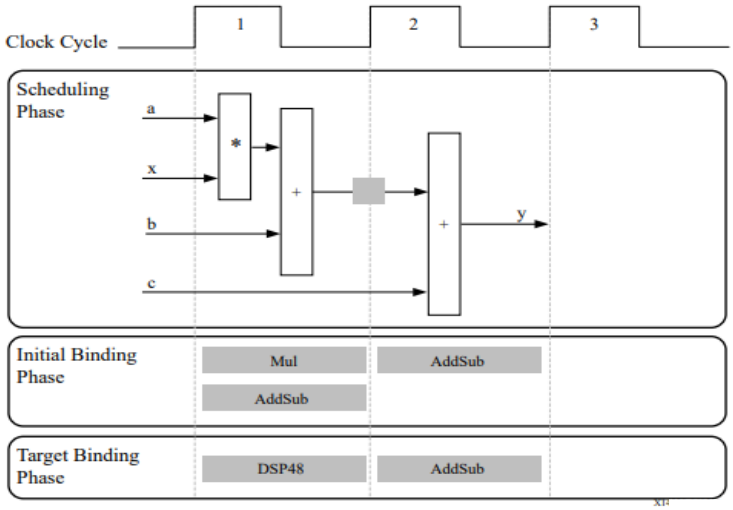

Figure 3.3: Scheduling block diagram for provided example [40]

Discussion:

In the above example, HLS will schedule the operation in such a way[40]:

1st clock cycle: Multiplication and first addition is performed (reads x, a and b)

2nd clock cycle: Second addition and output generation (reads c and generates output y)

Figure 3.3 shows the scheduling for the above example at each cycle. Many scheduling algorithms have been proposed in the past. The two simplest (i) As-soon-as-Possible (ASAP) and (ii) As-late-as-Possible (ALAP). As soon as possible algorithms also called the ASAP algorithm, as the name suggests it initiates all the operations at the earliest cycle based on data dependencies [31]. Here the successor node is executed only after the parent node has completed the computation. This algorithm results to be in the compaction of microcode [32]. As late as possible (ALAP) algorithm is complementary of the ASAP, i.e. it performs the operations during the latest cycle. Hence some of the operations are delayed to the next cycles [31].

Binding:

Binding refers to the association of operation and memory access to the hardware resource. Due to the dependency on hardware resources, device details play a major role as it provides

information on resource availability and can result in an optimal solution. Every operation is associated with one of the functions capable of performing the operation but in case there are multiple choices available the algorithm must opt for the optimized solution. Resource utilization is also optimized by sharing storage units among non-overlapping variables. Here connectivity binding also plays a major role as transfer between the components is bounded through connection units like bus or multiplexer, hence the assumptions regarding connectivity delay and area are made during the early stages of HLS [6].

In the above-provided example, arguments are input and output data ports. Operation is y = x*a+b+c; with each variable being a char type, according to which bit-width will be 8-bits. As the return type for the function is a 32-bit integer, the size of the output port would also be 32-bit [40]. Figure 3.3 shows the way binding would be performed for the operation. During the first clock cycle, the multiplication is being performed using a combinational multiplier (Mul) and the addition operation is performed using a combinational adder/subtractor (AddSub). According to the availability of resources HLS can implement multiplication and addition operation using DSP48 resource which is available in FPGA for the proper balance between high-performance and efficient implementation. Hence the number of resources required can be decreased to “DSP48” and “AddSub” instead of “Mul” and “AddSub” [40].

Generation:

After the decisions are made based on allocation, scheduling, and binding stages for the design, the RTL model is generated for the provided synthesized design.

The RTL model is generated using register-transfer components consisting of a controller and data path. A data path consists of various storage elements such as register, register files, and memories, functional units such as ALUs, multipliers, shifters, and interconnect elements such as tristate drivers, multiplexers, and buses [6].

All the components are allocated according to the required quantities by the operations and are interconnected through buses, depending on the type of component it can take one or more clock cycles to complete execution process. Execution of various components can also be pipelined

through the data path and controller stage. In the pipelining stage, there might be additional control signals from the controller to the data path to avoid violation of dependencies and status signals from the data path to the controller which keeps the controller updated regarding the data status whether being ready or not. The controller consists of various FSM to maintain stability in data transfer. Data inputs and outputs provided for the design are connected to the data path, and the control signals and outputs are routed to the controller [6].

3.3 Industrial Tool Used

As discussed in section 3.1, there are many commercial high-level synthesis tools available such as Xilinx Vivado HLS, Cadence Stratus, and CyberWorkBench provided by NEC Corporation. All these software are based on the C/C++/SystemC as their high-level language.

3.3.1 CyberWorkBench

In the provided work, the CyberWorkbench is used as the HLS tool to synthesis RTL from SystemC and Behavioural Description Language (BDL). It was launched by NEC Corporation. NEC has been developing a C-based behavioral synthesis called 'Cyber' and certain verification tools around it. All these functions are sub-part of one IDE tool called "CyberWorkBench" (CWB). The main Advantage is that CWB is based on the "All-in-C Synthesis" approach. BDL can be defined as a subset of ANSI C, as certain constructs are excluded because of being non-synthesizable and additionally certain hardware constructs included to synthesize RTL from the C program. ANSI C datatype help in optimizing the bit width of the variables. SystemC is a subset of C++ which provides better flexibility with more synthesizable constructs than BDL. In HLS, the major concern is that RTL becomes black box hence several verifications are done to the validation function of the design [33].

Figure 3.4 shows the design flow of CWB from C to HDL synthesis and at which stage verification of RTL is performed.

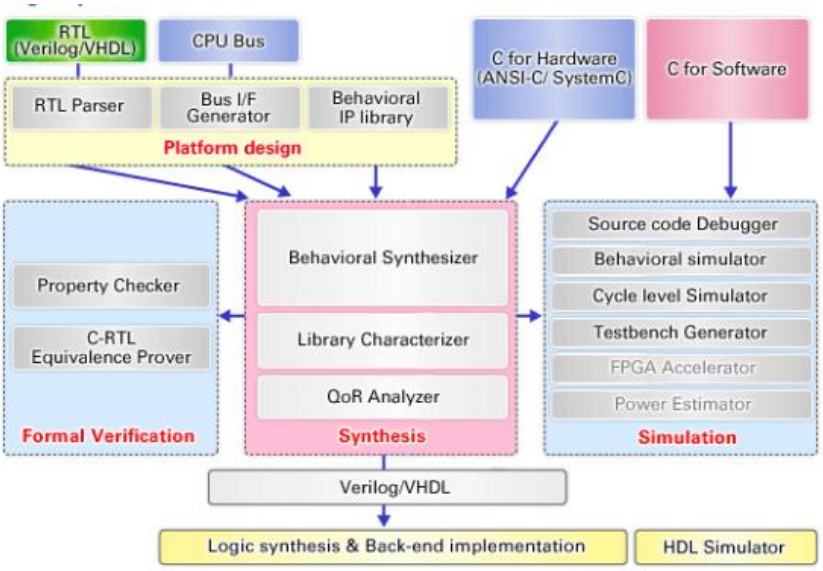

Figure 3.4: Design Flow of CyberWorkBench [41]

The major advantage of using CWB is it provides automatic and manual scheduling providing an upper hand to the designer. In manual scheduling, the designer can specify the number of clock cycles for a part of the code. In automatic scheduling, operations are scheduled by GUI and many optimization techniques may be used by GUI such as unrolling the loops and introducing intervals between the operations. In automatic scheduling, users can still manually schedule some blocks of the design using various pragmas. Base on the standard RTL synthesis tools, CWB generates technology libraries that are further used to generate functions and memory libraries for the design. CWB provides explorer to analyze and balance the design area and latency while considering various compiler settings and can optimize resource utilization [33].

CWB verifies different levels: behavioral, cycle-accurate simulation, and RTL. Among these cycles, accurate simulation is very important. During the cycle-accurate simulation input data are provided in the form of different files for each port. Cycle accurate simulation runs way faster compared to the RTL model and it allows designers to simulate and debug both hardware and software issues at the C/C++ level. Any required changes can be done directly at the C level and the entire simulation process is repeated [33].

In addition to verification, CWB generates a report showing a detailed analysis of power consumed by the design. A set of libraries are present for a different technology to provide accurate power estimation for the selected technology. A "QoR" report is also generated providing a descriptive analysis of design which includes area consumed, number of states, critical path delay, number of wires, and routability. This help in architecture exploration for the design to optimize the resource utilization or to decrease latency in the output obtained. CWB also provides flexibility to predefine constraints (such as area, latency, power) at C level and synthesize RTL with the required design specifications [33].

3.3.2 Higher-Level Language Utilized

In the presented work, SystemC is used as a high-level language. SystemC utilizes C++ standards for creating an architecture for the design. SystemC provides an upper hand over C language by including additional features like datatype with changeable bit width. This leads to an optimized design than in C/C++ language which offers datatype with a fixed bit width [34].

SystemC offers various library which provides support to various modeling features such as [34] :

- Similar to modules in Verilog, it lets decomposition of the entire system into modules and sub-module. Each module instantiated with another module to form a hierarchy for the entire design.
- It allows to connect different module with the help of channels. These channels are connected using ports and exports.
- It helps in scheduling and synchronization between the processes with the help of events and sensitivity. The events are executed with the help of an in-build sc_event class. 2 types of sensitivity are offered in SystemC, static, and dynamic. Static sensitivity is performed using commands such as SC_THREAD or SC_METHOD at the beginning of the simulation. Dynamic sensitivity enables runtime sensitivity change.
- Hardware-focused datatypes are present in the SystemC libraries. These are similar to the Verilog datatypes often used for hardware modelling. These datatypes assist in modeling complex algorithms into the design flow.

3.3.2.1 SystemC datatypes

All the data types used in C++ are supported with SystemC libraries. Additionally, SystemC provides a type class with “sc_datatype” to represent bit width for the register being synthesized as a part of the hardware. These datatypes offer limited precision [35].

SystemC datatype classes can be categorized as [35] :

- Limited-precision integers – These classes are derived from base class from class such as sc_int_base or sc_uint_base. These represent signed or unsigned integer with precision set up to certain value limited-precision integer by referring to their native C++ representation and specified bit width.
- Finite-precision integers - These classes are derived from base class such as sc_signed or sc_unsigned. These represent signed or unsigned integer with precision set up to certain value limited-precision integer by referring to their specified bit width.
- Finite-precision fixed-point types - These classes are derived from base class such as sc_fxnum. These represent signed or unsigned fixed-point number with precision set by specified bit width, bits allocated to integer representation, quantization mode, and overflow mode.
- Limited-precision fixed-point types – These classes are derived from base class such as sc_fxnum_fast. These represent signed or unsigned fixed-point number with precision set by native C++ floating-point representation and the specified bit width, bits allocated to integer representation, quantization mode, and overflow mode.
- Variable-precision fixed-point type - These classes are derived from base class such as sc_fxval. These represent a fixed-point number with changeable precision and does not take into consideration quantization or overflow.
- Limited variable-precision fixed-point type - These classes are derived from base class such as sc_fxval_fast. These represent a fixed-point number with precision set by native C++ floating-point representation and with changeable precision without considering quantization or overflow.

- Single-bit logic – This logic datatype takes into account four states/logic, 1, 0, high impedance (‘X’), and unknown state (‘Z’). These are assigned as literals for single-bit logic.
- Bit vectors - These classes are derived from base class sc_bv_base. These implement multiple data types with each bit consisting of either logic 0 or logic 1.
- Logic vectors - These classes are derived from base class sc_bv_base. These implement multiple-bit data type with each bit consisting of either of logic 0 or logic 1 or high impedance (‘X’) or unknown state (‘Z’)

In the presented work, a limited number of data types are being used. Hence only those data types are discussed as below:

- sc_in – This class template is used to model an input port to a module. Within this data type for the port is specified.
  For Example: sc_in< sc_int<32> > var
  Here the input port is a signed integer of size 32 bits
- sc_out – This class template is used to model an output port to a module. Within this data type for the port is specified.
  For Example: sc_in< sc_int<32> > var
  Here the output port is a signed integer of size 32 bits
- *sc_int* – This class template is used to represent signed integer with specified word length. A template argument is used to specify word length.
  Template example:
  sc_int<W> variable;

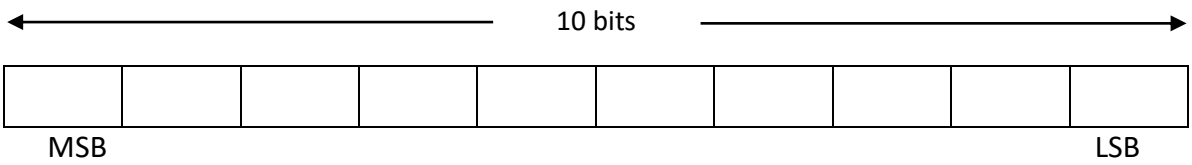

Figure 3.5: Bit Allocation for sc_int template

Here W signifies a bit width of variables. In this case, W is set to 10 then 10 bits are allocated to the integer part of the variable. Out of these 10 bits, 1 bit (MSB) is used to signify a sign of the integer (whether positive or negative integer)

Maximum value it can store = 1023

- *sc_fixed* – This class template represents a signed finite-precision fixed-point value.

  Template example:

  sc_fixed<wl, iwl, q_mode, o_mode, n_bits> filter

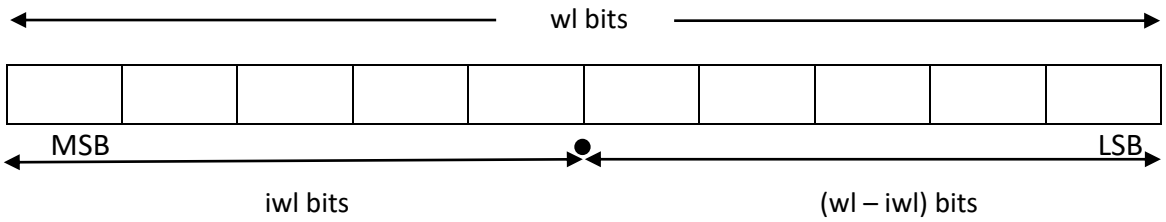

Figure 3.6: Bit Allocation for sc_fixed template

Parameter description:

- wl = Defines word length for the variable
- iwl = Defines the number of bits being allocated to the integer part of a variable
- q_mode = Defines the mode of quantization
- o_mode = Defines the mode of overflow
- n_bits = This defines the number of bits being saturated

CHAPTER 4

FIELD PROGRAMMABLE GATE ARRAY

4.1 Introduction

An FPGA is an integrated circuit (IC) with the capability of being reprogrammed for different algorithms after fabrication. This gives an upper hand to FPGA over ASIC chips which even though provide high throughput, can only be designed for specific tasks. Also, ASIC design usually takes considerable longer time to design (from months to a year) and cost millions of dollars to develop a prototype whereas FPGA can be configured within seconds and cost between hundreds to thousands of dollars [36]. Hence, due to their ability to reconfigure and being cost-effective, they are preferred to validate various devices before passing them through the silicon development stage.

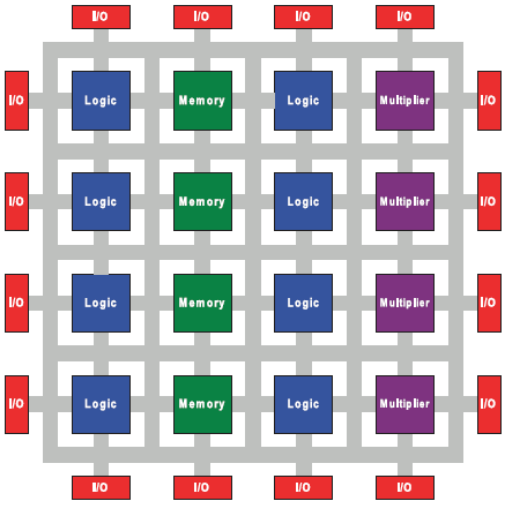

Figure 4.1: Blocks Diagram of FPGA [36]

Figure 4.1: shows a block view of basic FPGA structure and the components embedded within it.

As shown in Figure 4.1, FPGA blocks consist of [36]:

- Programmable logic blocks
- Predefined logic blocks (multipliers and memory blocks)
- Routing fabrics, and
- Input/Output Pads

As shown in Figure 4.2, the programmable logic blocks are usually made up of look-up table (LUT), flip-flops (FF) and multiplexer. Inputs of the LUT vary in number from 3 to 8. This is decided based on experiments conducted. Currently, adaptive LUTs are used which implement 2 functions per single LUT [37]. These programmable logic blocks are responsible to perform all the basic to complex computation and at the same time act as a storage element with the help of memory blocks interconnected with them. The interconnections are performed with the help of routing fabrics. Input/Output pads are used to connect any external device to the fabric of FPGA. Fig 4.1 shows the basic block diagram of FPGA. With the growing application of FPGA, certain additional blocks are required to perform specific functions. Hence, heterogeneity is introduced by embedding blocks such as memory blocks or multipliers [36].

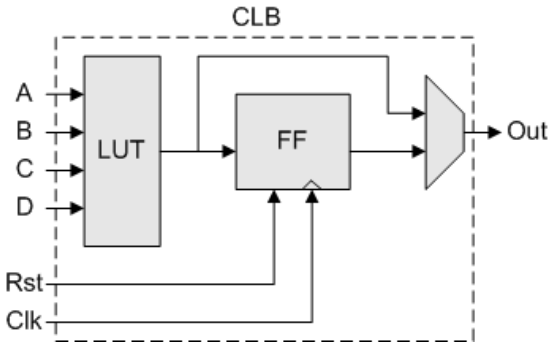

Figure 4.2: Structure of Logic Block [38]

For different applications, memory needs to be configured differently. Hence memory blocks need to be flexible and must be configurable. As the designs keep on becoming more and more complex and consume more memory, the size of memory required also keeps increasing. Hence memory has become an important part of the system and covers a significant fraction of the die area [36].

Even microprocessors are often used in conjunction with FPGA logic which provides flexibility, low power utilization, and bandwidth communication between the processor and FPGA [36].

4.2 Commercial Logic Block

One of the most popular commercial logic block used in Altera based FPGAs is ALM. It has the ability to be partitioned effectively into smaller LUTs. The advantage of introducing ALM is that it offers 1.8 times density over the other commercial logic blocks. The basic structure consists of combinational logic, 2 registers, and 2 adders [42].

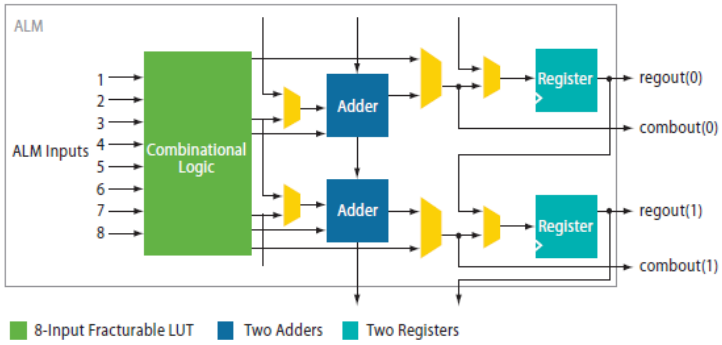

Figure 4.3: Block Diagram of ALM [42]

As shown in Figure 4.3, there are 8 inputs to the combination logic block and consists of LUT which is further divided into 2 adaptive LUTs (ALUTs). As the LUTs are constructed using SRAM bits, they can hold memory and multiplexers to help in the selection of bits to drive the output. Incase larger LUTs are required to be built, smaller LUTs are used to design them. But this might result in varying area consumption and delay in the circuit. Various research was conducted for the same, and it was observed that a basic 6-LUT could result in a 14% improvement in the performance by decreasing the number of levels in the critical path. But this also resulted in a penalty of 17% in the area [42].

FPGA Prototyping boards

In this thesis, we use a DE1-SoC Board that mounts an Intel Cyclone V SoC FPGA. This type of newer FPGAs contain embedded processors and reconfigurable fabric on the same devices, in this case a dual core ARM-Cortex M.

Some of the salient features of the DE1-SoC board are: [43]:

- It has Altera Cyclone® V SE 5CSEMA5F31C6N device
- It is programmed in JTAG Mode using USB-Blaster
- It has 4 push-buttons on the board
- It has 10 slide switches
- It consists of six 7-segment displays
- The board has 10 red-colored LEDs connected to FPGA chip
- It has onboard VGA DAC (high-speed triple DACs with the width of 8-bits) with a VGA-out connector

Figure 4.4 provides an overview of the board with all the peripherals as well as the FPGA mounted on it. One additional advantage of this board is that it is very inexpensive.

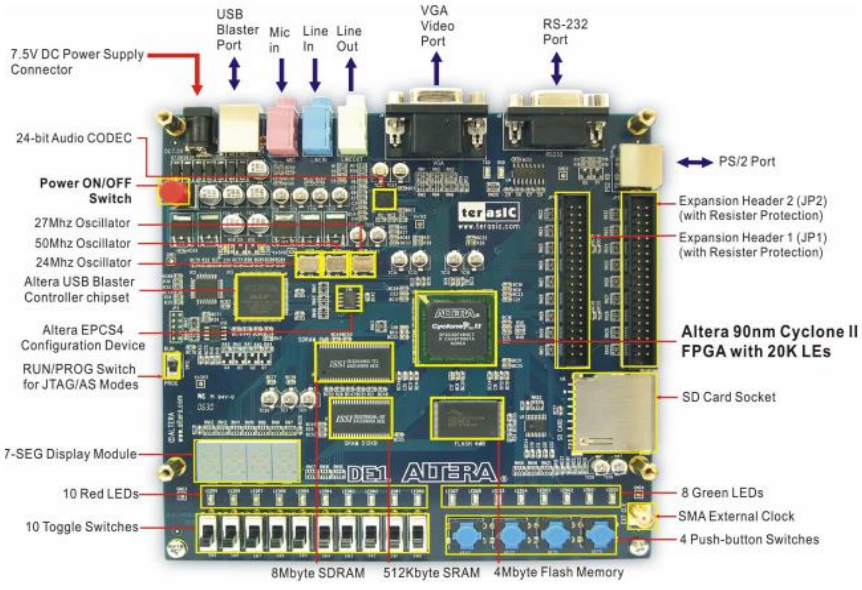

Figure 4.4: Overview of DE1-SoC Board with the peripherals [43]

CHAPTER 5

FPGA ACCELERATION OF CONVOLUTIONAL NEURAL NETWORK

5.1 Focus of Presented Work

This chapter summarizes the main body of work done in this thesis. It presented the CNN architecture developed to identify the image of handwritten digits from 0-9 using the MNIST dataset.

The MNIST dataset was designed by the National Institute of Standard and Technology (NIST) consisting of binary images of handwritten digits. The dataset consists of 60,000 images for training and 10,000 images for testing, and each image with a pixel size of 28x28 with 256 grey level. These handwritten digits are from 500 different writers [45]. The dataset is available online and is considered a standard dataset for developing DL model to identify handwritten digits. The reason behind targeting such a dataset is that a smaller number of stages are required in the architecture. Since these are binary images, the pattern can be recognized with a limited number of feature/filters maps unlike the other application (face identification where many features need to be extracted). This gives us more time to focus on developing and implementing each layer of CNN on the FPGA. The dataset used here is in the form of an array of numbers. Additionally, the robustness of architecture is analysed here by breaking neurons at various stages.

To verify the performance of the CNN on the FPGA, a similar software model was developed in C++, where weights were also trained. The software model was developed to validate the performance of the architecture developed. Once validated, a similar hardware model was developed and trained, and the accuracy of the model was observed.

5.2 Architecture Flow and Data Preparation

5.2.1 Pre-processing of Dataset

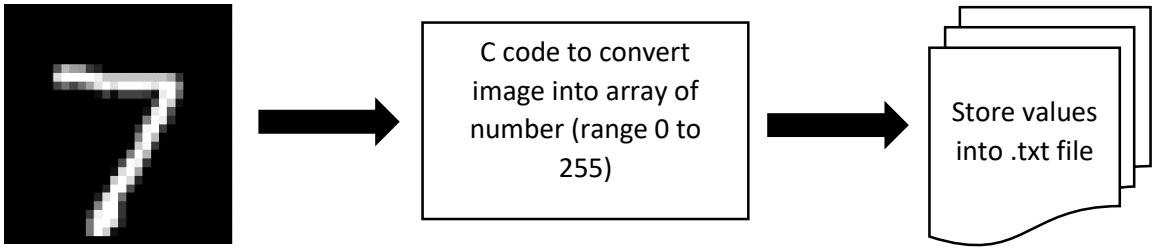

Figure 5.1: Converting image into an array of numbers

Figure 5.1 shows how an image is converted into numeric values. First, the image needs to be converted into an array of numbers. The range of numbers is obtained for the input array is between 0 to 255 (as the input images are greyscaled). Hence, it makes the calculation complex by introducing more deviation in the values. Therefore, the numbers are scale down to the range of -0.5 to 0.5 to make them 0 centric which the training task much simple.

arr_new = (arr/255) – 0.5;

where,

arr = input data in raw (unmodified) form

arr_new = modified input data obtained after downscaling the original data

5.2.2 CNN architecture implemented

Once the data is normalized to -0.5 to 0.5, it is passed through a sequential model represented in the form of a linear stack of layers, consisting of first convolution layer, pooling layer, fully connected and softmax classification. Figure 5.2 shows the overall flow. As discussed in the CNN chapter, the convolution layer consists of filters that slide through an input data array. Here we are using 1 filter and the parameters are trained accordingly. Then data is passed through the max-pooling layer with stride 2 hence the number of elements after this stage becomes half than that of the previous one.

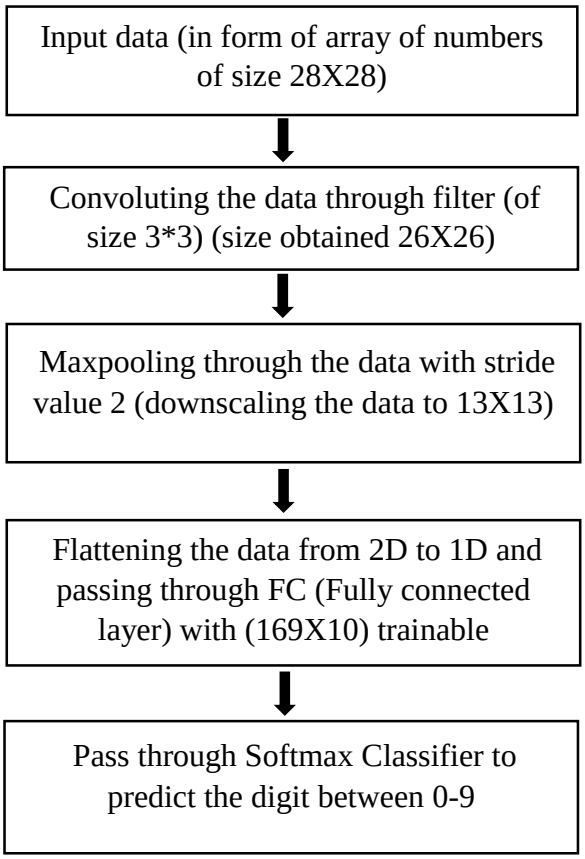

Figure 5.2: Flow of CNN Architecture

Then the data is flattened, i.e. 2D array of size (28X28) is converted into a 1D array of 169 elements. This is further passed through a fully connected layer with 169X10 features. Here, each element of the max-pool layer is connected to 10 neurons i.e. 10 features. Hence the feature map consists of (13x13x10).

This is again passed through the activation layer for classification, which is softmax classification here. The classifier gives the probability for each class. Based on the prediction, the algorithm decides whether to calculate the loss and train the feature map.

5.3 Implementation of Software and Hardware Model

5.3.1 Software model implementation

The software model is developed in C++ language and compiled using g++ compiler. As the hardware architecture is developed in SystemC, the software model is used as the reference model and modified to synthesizable SystemC including its datatypes. Because the description has to be synthesizable, no built-in libraries can be used as the HLS tools need to generate the hardware for them. HLS is then used to generate the RTL code.

The CNN description developed in this work is similar to the one implemented using the original python code. The input data of the array is passed through the convolution layer, then through the pooling layer, and then through the dense layer with activation function for classification between 10 classes. In the dataset along with the image data, labels for each image are also provided to perform supervised learning and test the prediction of the model.

To train the weights, gradient descent is used and also regularization is done to obtain an optimal solution. The next subsection describes this in detail.

5.3.1.1 Gradient descent

Gradient descent is one of the most popular learning algorithms used in ML (Machine Learning) and Deep Learning (DL) model. It is a partial derivative of a set of features with respect to the input data. Gradient descent is a convex function and finds the value which helps in reaching the global cost minimum [46]. Hence gradient is directly proportional to the steepness of the curve. Figure 5.3 shows the plot between change in weight and how it affects the cost function.

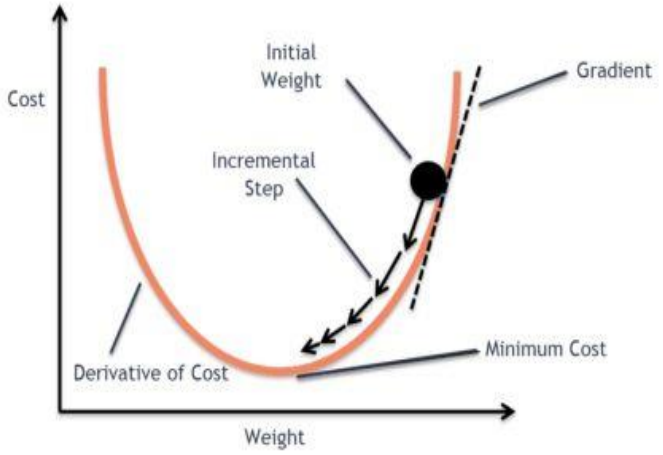

Figure 5.3: Plot between Weight and Cost Function [47]

The standard formula for gradient descent is,

$$\theta = \theta - \alpha * \nabla [J(\theta)]$$

Here,

θ is the Feature being trained

$J(\theta)$ is the loss function with respect to parameter targeted

Here alpha is learning rate, which plays a crucial role while training the model in ML

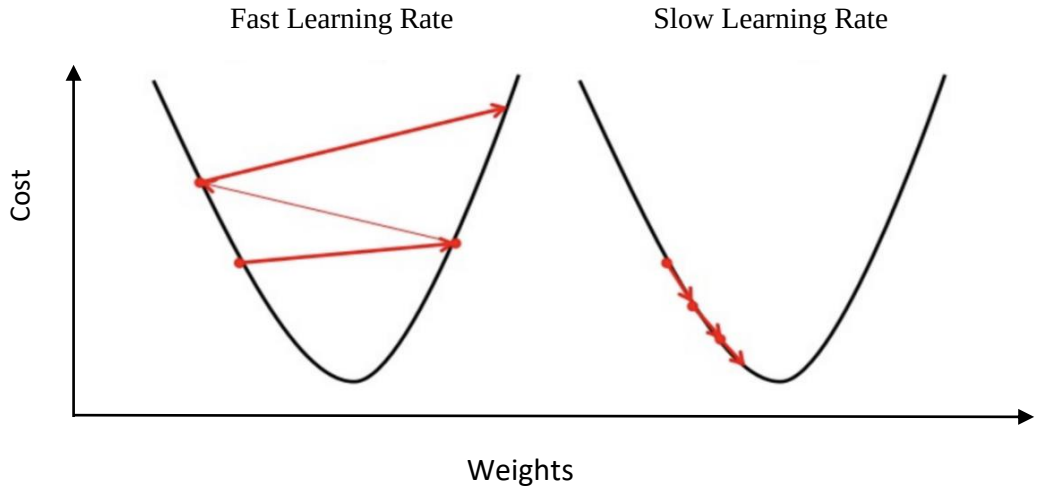

Figure 5.4: Plot showing the effect of learning rate on the Cost function [48]

Figure 5.4 shows the effect of learning on the cost function, in case the learning rate is too fast i.e. too high, the gradient might miss the global minima whereas if the learning rate is too slow i.e. very small in value, it might take too many iterations even with a huge dataset to train the model.

In the presented work we have used a slower (smaller) learning rate to compensate for the variance that occurred due to the noisy datasets. In the present design, the constant learning rate helps in converging towards a global minimum at a medium pace. An alternate method can be assigning learning rates based on accuracy obtained and may increase or decrease convergence based on loss obtained. But while using the method, it brought further complexities and resulted in overfitting of data as there is some noisy data present which causes overtraining of data.

5.3.1.2 Overfitting

During the training of model features, it is iterated across a huge dataset. The purpose of training the features is to minimize the overall loss and at the same time fitting the model into all the points of the dataset which might include catching of noisy data which might affect the aim of creating a generalized model [49]. Figure 5.5 shows the way features would fit the model while training them though a large dataset.

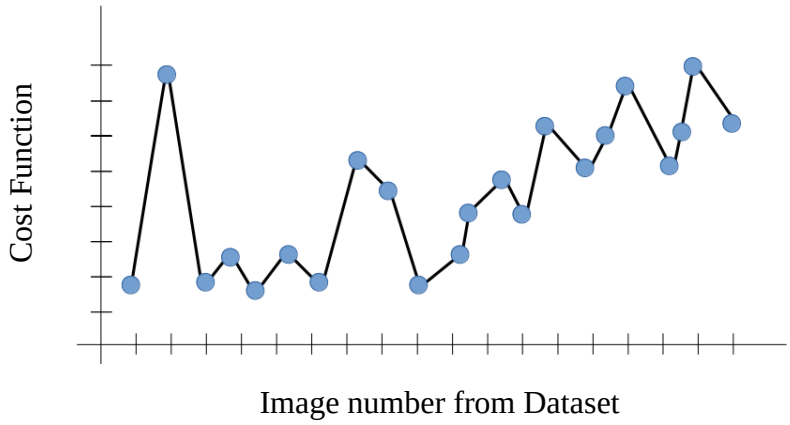

Figure 5.5: Illustrates the way features would train to fit the model into the target points [49]

Whereas the target is to capture the dominant trend as given in Figure 5.6.

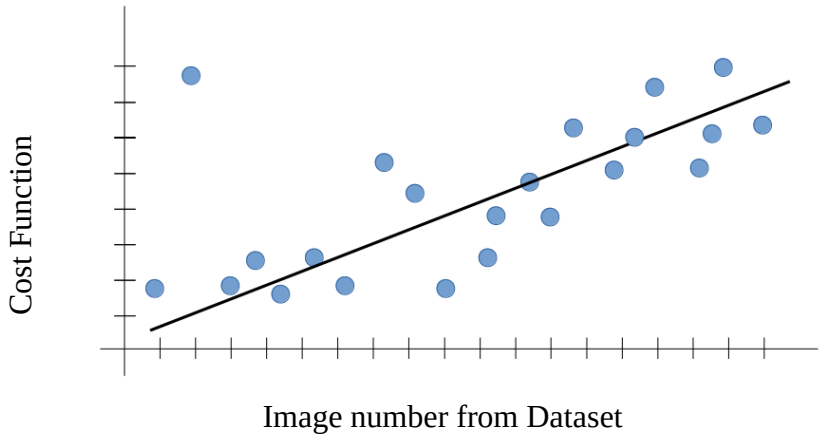

Figure 5.6: Way features need to be trained to develop a generalized model [49]

In case the dominant trend has not captured the model won't be able to recognize images with similar patterns which would result in disregarding the purpose of ML.

To avoid overfitting of weights due to noisy dataset regularization methods are used. The regularization methods are used to stabilize the situation in case the line trying to capture the dominant trend deviates from the trend. As discussed previously there are two types of regularization methods and in the presented work L1 regularization is used. The value of the regularization parameter is chosen to be small but can be varied based on the application and amount of compensation required. In the presented work various values were used and the optimum value of 0.375 was chosen.

5.3.2 Hardware Implementation

The hardware implementation is divided into 2 parts:

1) Training part
2) Network instantiation part

During the training part the new weights are being set by the embedded ARM processor until the exit criteria is reached, while the actual CNN architecture is implemented on the FPGA. The ARM processor connects with the FPGA through the on-chip AXI bus. This allows to update the feature maps that are passed to the CNN for evaluation.

The images are present in the form of an array of numbers here. Each element of the array is passed every clock cycle to the CNN. The ARM processor uses the AXI bridge to communicate with the CNN. The AXI bridge is further interconnected to the Avalon MM interface to perform read and write operations with master and slave interfaces as shown in Figure 5.7. Here the ARM processor works as the master and FPGA works as a slave through the implementation [50].

There are 3 types of AXI bridge interfacing parameters [50]:

FPGA-to-HPS interface width – This parameter is used to enable or disable FPGA-to-HPS interface and in case enabled, data width selection to 32, 64 or 128 bits

HPS-to-FPGA interface width - This parameter is used to enable or disable HPS-to-FPGA interface and in case enabled, data width selection to 32, 64 or 128 bits

Lightweight HPS-to-FPGA interface width – This parameter enables or disables the lightweight HPS-to-FPGA interface and in case it is enabled, the default data width is 32 bits

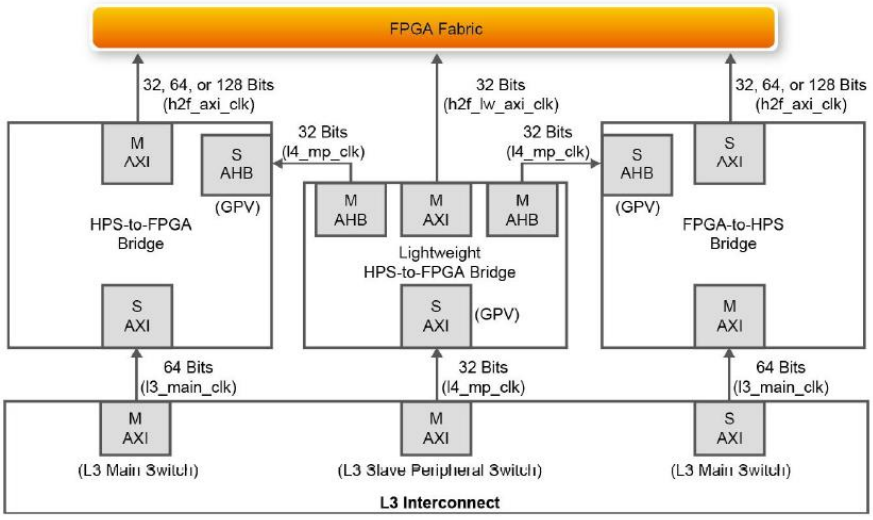

Figure 5.7: Block Diagram of AXI Interfacing with FPGA Chip

In the proposed work, interfacing is done using the Lightweight HPS-to-FPGA AXI bridge. This interfacing is done using one of the tools provided by Quartus Prime called Platform designer (also known as Qsys). While interfacing Avalon-MM certain signals types are included according to establish the required communication. Various signals are incorporated to read and write data and perform proper synchronization of the data.

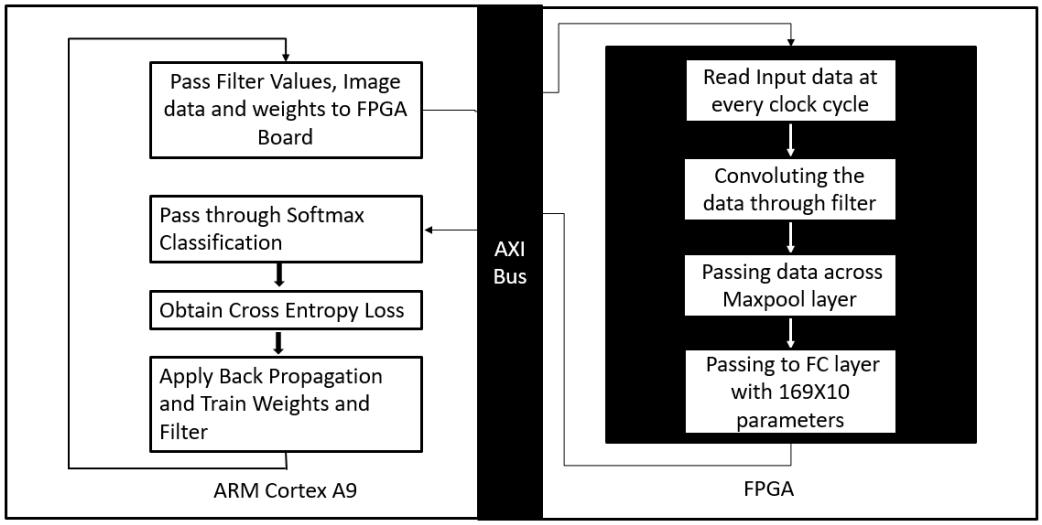

Figure 5.8: Block Diagram of CNN Implementation on DE1-SoC Board

Software languages offer floating-point data type which eases the work of bits allocation to fractional and integer part but the size of the variable remains fixed to 32 bit whereas, for the hardware model developed in SystemC, the fixed point data type is used. Fixed point data type gives the liberty to limit the bit width of register according to requirement hence the area of CNN can be optimized. As discussed previously in the HLS chapter, sc_fixed is the fixed-point data type used here to perform all the fractional intensive computations.

Template for sc_fixed is,

sc_fixed< X, Y, q_mode, o_mode, n_bits>

Here,

X = bitwidth of variable

Y =Number of bits allocated to integer part

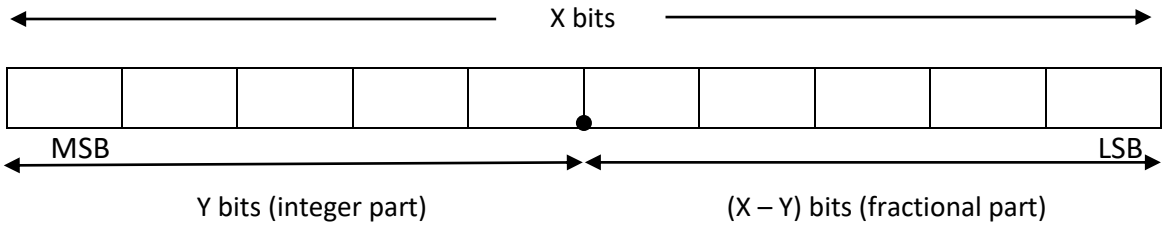

Figure 5.9: Bit allocation for sc_fixed template

Depending on the size or bit width of the register, resource utilization also varies and so does the precision. As the fewer bits are allocated to the fractional part, the precision of the register changes and so does the accuracy of the entire model. For example, let's say X is 20 and Y is 10. Then the precision can be obtained up to 0.000976562 (2^ ((-1) *(20-10))).

Depending on the number of bits allocated to fractional part in the feature maps the models were named, and accordingly bits are allocated to the internal registers. Then the accuracy of all the models is examined and compared to obtain the optimized design.

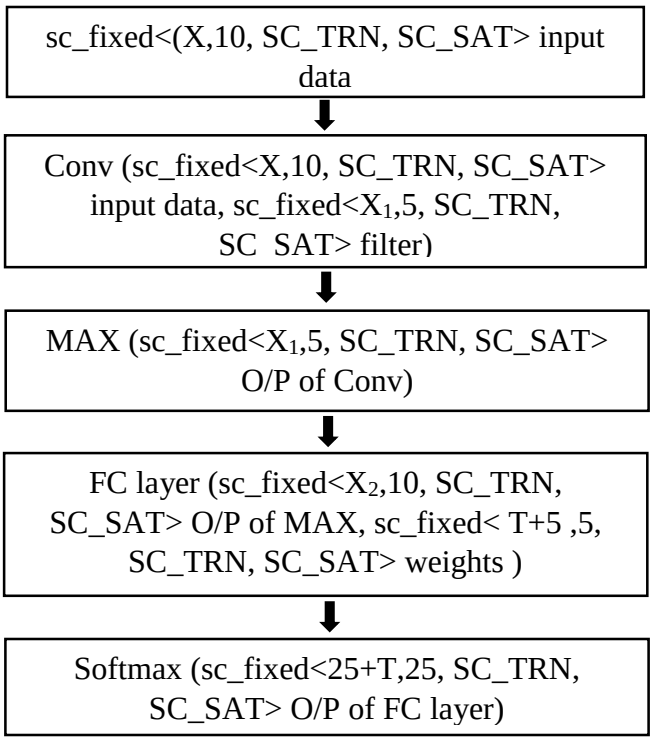

Figure 5.10: Base Design flow for all Model

In the block diagram shown in Figure 5.10, parameters X, X_1, X_2 and T are varied according to the model. Table 5.1 provide values for these parameters according to the model proposed. Parameter T signifies the precision up to which weights are being trained. Hence the models are named according to the value allotted to parameter T.

Five different bit-width models were developed and compared as follows:

- 17 Fractional Bit Width
- 12 Fractional Bit Width
- 9 Fractional Bit Width
- 5 Fractional Bit Width
- 3 Fractional Bit Width

Table 5.1: Values allocated to the Parameters in the Base Model

Model	**X bits**	**X_1 bits**	**X_2 bits**	**T bits**
17 Fractional Bit Width	27	20	27	17
12 Fractional Bit Width	22	17	22	12
9 Fractional Bit Width	20	15	21	9
5 Fractional Bit Width	16	11	15	5
3 Fractional Bit Width	14	9	13	3

Further, a detailed study was performed on the resource utilization by each model.

Using different Bitwidths leads to CNNs implementations of the different areas as the different number of LUTs and registers are used. Table 5.2 summarizes the synthesis results for the CNN implementations shown in table 5.1

Table 5.2: Resource Utilization Summary by Each Model

Parameters	**17 Fractional Bit Width**	**12 Fractional Bit Width**	**9 Fractional Bit Width**	**5 Fractional Bit Width**	**3 Fractional Bit Width**
Logic utilization (in ALMs) (Total 32,070)	12,997 (41 %)	12,622 (39 %)	12,634 (39 %)	12,502 (39 %)	12,465 (39 %)
Total registers	6043	5766	5552	5006	4980
Total pins (Total 457)	368 (81 %)	368 (81 %)	368 (81 %)	368 (81 %)	368 (81 %)
Total block memory bits (Total 4,065,280)	660,068 (16 %)	648,510 (16 %)	631,854 (16 %)	603,166 (15 %)	605,546 (15 %)
Total DSP Blocks (Total 87)	1 (1%)	2 (2 %)	1 (1 %)	1 (1%)	1 (1%)
Total RAM Blocks (Total 397)	88 (22%)	86 (22 %)	83 (21 %)	82 (21 %)	80 (20 %)
Total DLLs (Total 4)	1 (25%)	1 (25 %)	1 (25 %)	1 (25%)	1 (25%)

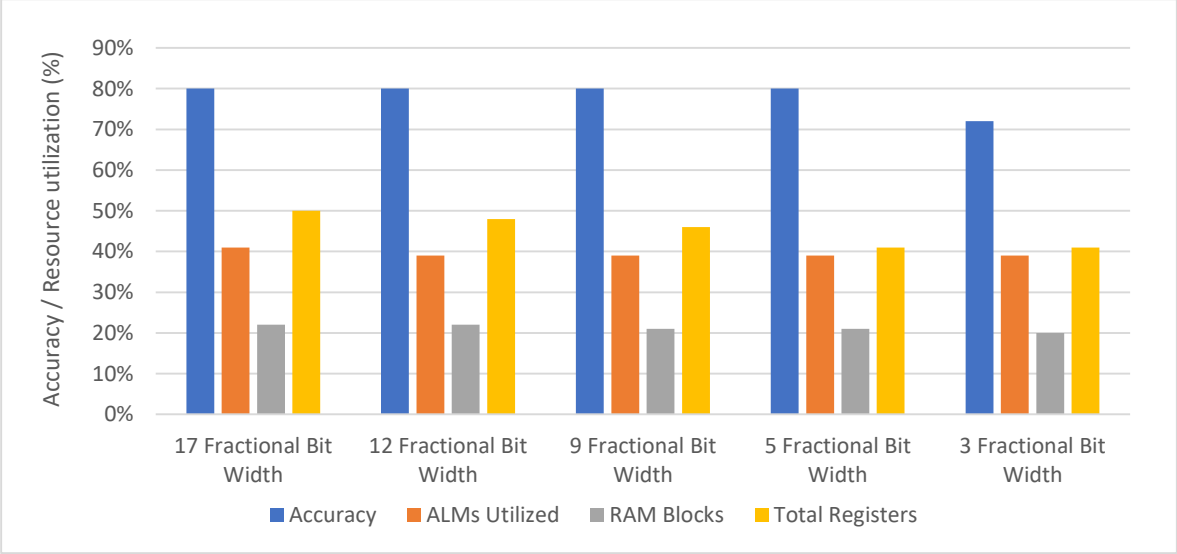

Figure 5.11: Resource utilization and Accuracy comparison between various models

From the results in Figure 5.11 and Table 5.2, it can be observed that with decreasing bit width, the utilization of ALMs, registers, and memory blocks is also decreasing. Surprisingly, decreasing the bit width the network accuracy is still maintained to 80% till the 5-Fractional bit width model. On decreasing even further, the accuracy drops drastically. Hence the 5- Fractional bit width model is the most optimized model observed here. It should be noted that for every bitwidth the CNN is fully re-trained.

These experiments show that CNNs have large potential for optimizations when mapped on hardware as opposed to running them on SW of GPUs as they do not require floating-point arithmetic.

One additional study that is investigated is how adding the in-situ training circuitry to the FPGA affects the total area of the design. Traditionally, ANNs are trained offline and the ANN hard-coded in the circuit, hence no training circuitry is needed. It is, therefore, necessary to study the overhead of our design that allows the in-situ re-training of the CNN vs. the traditional hard-coded version.

Table 5.3 and Figure 5.12 show the results of the two versions. It can be observed that giving the CNN the flexibility to re-train itself leads to a significant area overhead (41% ALMs vs. 155 ALMs). Considering that FPGA can be re-programmed in the Field raises the valid question of the need for in-situ retraining. One advantage of in-situ retraining is that it allows to re-train the particular FPGA when a fault happen, e.g. a wire break. This cannot be done offline as the fault is unknown to the outside world.

Table 5.3: Resource Utilization Comparison between trainable weight and fixed weight model

	Trainable Weights	**Fixed Weights**
Logic utilization (in ALMs) (Total 32,070)	12,997 (41 %)	4,774 (15 %)
Total registers	6043	3807
Total pins (Total 457)	368 (81 %)	368 (81 %)
Total block memory bits (Total 4,065,280)	660,068 (16 %)	647,900 (16%)
Total DSP Blocks (Total 87)	1 (1%)	1 (1%)
Total RAM Blocks (Total 397)	88 (22%)	87 (22%)
Total DLLs (Total 4)	1 (25%)	1 (25%)

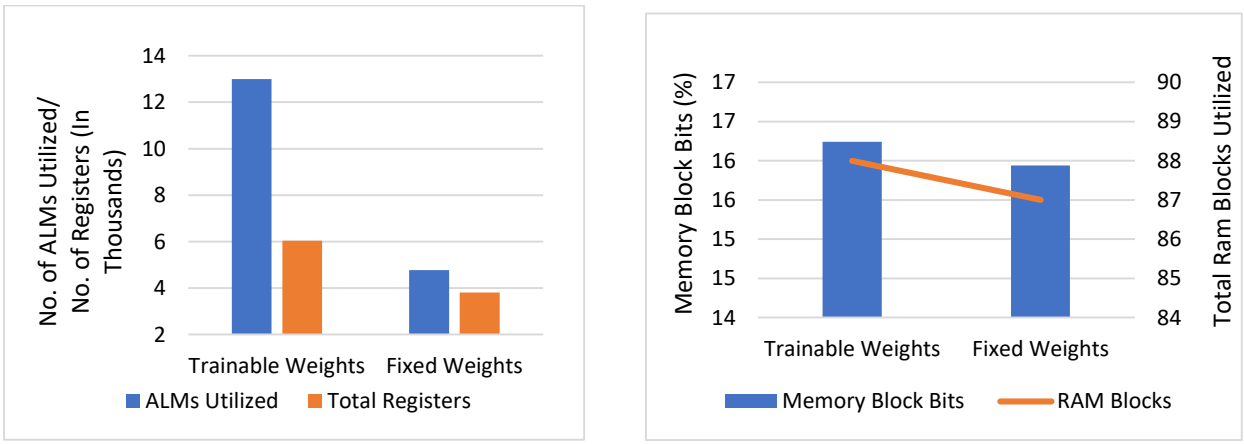

Figure 5.12: Resource utilization comparison between trainable weight and fixed weight model

5.3.2.1 Analyzing Robustness of CNN

The final part of this work deals with the study of the robustness of the CNN when the hardware "breaks". With the advent of technology, transistors are scaling to under 10nm making them more susceptible to permanent faults. It is therefore paramount to investigate how the accuracy of these CNNs gets affected when faults at different part of the CNN happen. When e.g. a connections breaks we can model this is a stuck-at-0 or stuck-at-1, the hardware might fail or produce unexpected results. Hence in the presented work, the robustness of the architecture is analyzed by breaking neuron connections and neurons at various stages to see the effect on the accuracy of the design. Based on the analysis, it can be concluded whether the design is efficient enough to reproduce similar results in the case of certain parts of hardware failure? And in case of results cannot be reproduced, can retraining of the parameters help in restoring the accuracy enough to a level that hardware need not be replaced?

Here neuron connections are broken at various stages to analyze the robustness of the CNN architecture. This is further divided into 2 part:

1. Breaking neuron connections between FC layer and classification layer
2. Breaking neurons in the max pool layer

1. Breaking the neuron connection between FC layer and classification layer

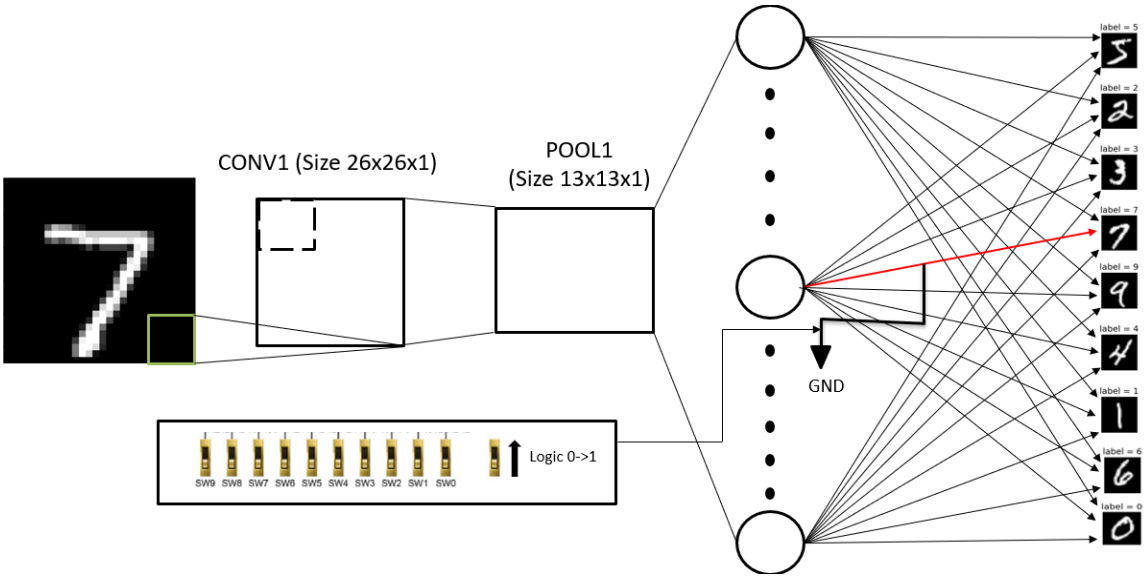

Figure 5.13: Architecture showing where the neurons are broken

Figure 5.13 conveys that here the connections are broken with the help of switches present on the FPGA board. Using these switches neurons inter-connections are broken between the FC layer and classifier for that class. Then the accuracy was analyzed.

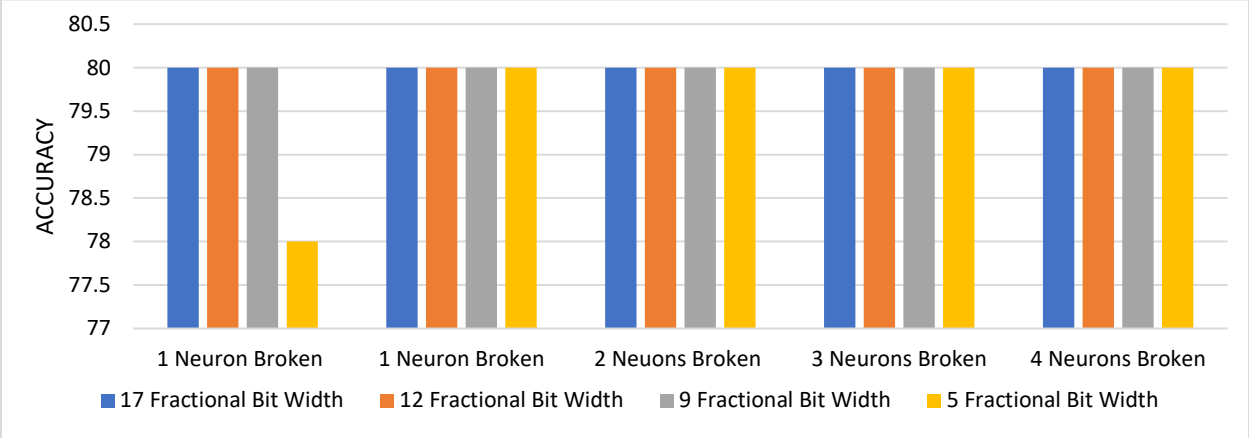

Figure 5.14: (a) Plot comparing obtained accuracy on breaking neuron connections in all models

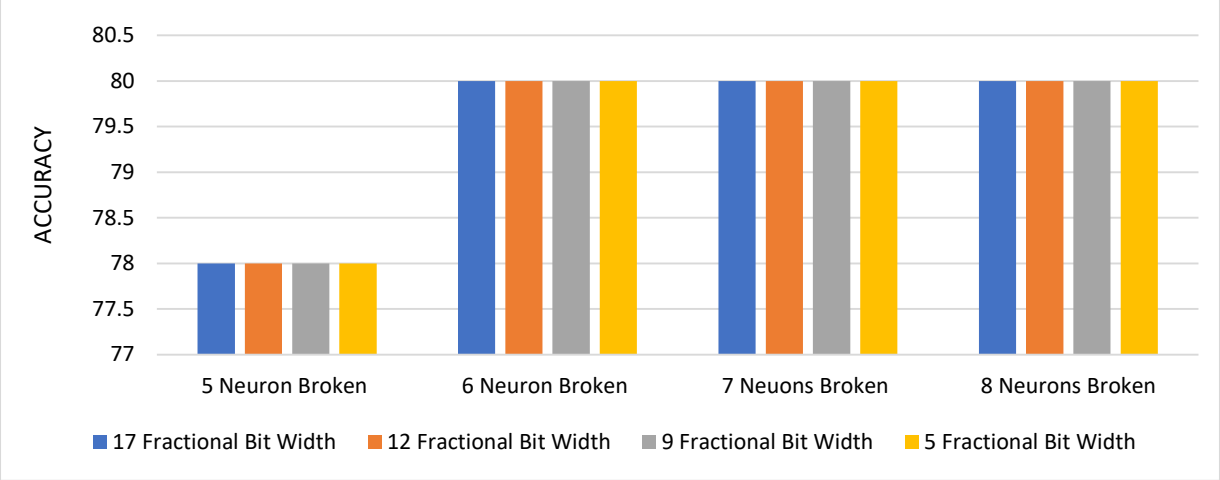

Figure 5.14: (b) Plot comparing obtained accuracy on breaking neurons connections in all models

Table 5.4: (a) Showing accuracy obtained on breaking neurons in various models

Neurons Broken	**Accuracy**			
	17 Fractional Bit Width	12 Fractional Bit Width	9 Fractional Bit Width	5 Fractional Bit Width
1	80%	80%	80%	78%
1 (Diff. neuron)	80%	80%	80%	80%
2	80%	80%	80%	80%
3	80%	80%	80%	80%
4	80%	80%	80%	80%

Table 5.4: (b) Showing accuracy obtained on breaking neurons in various models

Number of Neurons Broken	**Accuracy**			
	17 Fractional Bit Width	12 Fractional Bit Width	9 Fractional Bit Width	5 Fractional Bit Width
5	78%	78%	78%	78%
6	80%	80%	80%	80%
7	80%	80%	80%	80%
8	80%	80%	80%	80%

Figure 5.14 shows the comparison of accuracy obtained on breaking different numbers of neurons between the FC layer and the classification layer and how it affects the capability of the model. The results convey that breaking the connections between neurons in the FC and classification layer doesn't affect the accuracy of the model significantly. Here a different number of connections ranging from 1 connection to 8 connections were broken and the accuracy was obtained. Looking at the accuracy in Table 5.4, it can be observed that accuracy remains stable to 80 %.

Also, with a decrease in bit width, accuracy is maintained. This is due to the self-adjusting capability of the parameters. With the change in the model, the precision of the parameters also

changes. For example, in the 17- Fractional bit width model, weights were trained with the precision of $7.63X10^{-6}$ whereas for 5-Fractional bit width the weights were trained with the precision of $3.125X10^{-2}$. This brings a significant change while training the weights. In a higher bit width model, since the precision is better thus, the lower loss would also be taken into consideration whereas in the lower bit width model only significant loss would be taken into consideration. Hence, a lower bit width model might also result in avoiding overfitting of weights and would result in a better-generalized model. But in the case where the bit width is reduced very significantly, it might result in considering only high loss. This would further slow-down the learning process and wouldn't reach the global minima. For example, in a 3- Fractional bit width model that would train with the precision of 0.125, the accuracy obtained is 72% because the gradient would only be trained when the loss is very high. When loss is small but considerable, the weights wouldn't be trained hence, it wouldn't reach global minima even after increasing iteration or it might skip the global minima.

Though the accuracy of the model was not significantly affected on breaking the connection between neurons from FC to the output layer, the neuron did create a minor change in accuracy. On breaking 2 connections individually from FC to output layer for number 7, change in accuracy by which the model predicts the same image was observed. Here, three different images representing number 7 were taken into consideration. The data for the three different images has been recorded in Table 5.5 and Figure 5.15.

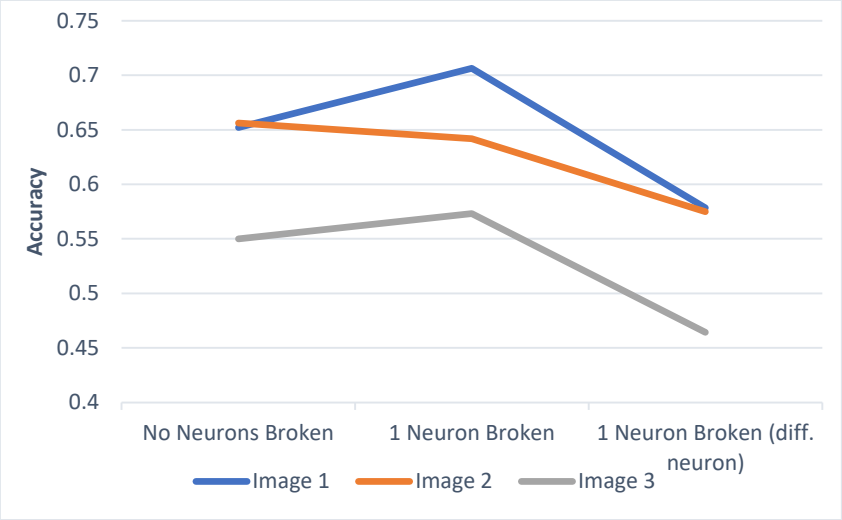

Figure 5.15: Change in prediction on breaking different neurons

Table 5.5: Showing Prediction obtained on breaking different neurons

Image Number	**No Neuron Broken**	**1 Neuron Broken**	**1 Neuron Broken (diff. neuron)**
1	0.652099	0.706465	0.578295
2	0.656052	0.641815	0.574932
3	0.550084	0.573161	0.464376

From Figure 5.15 and Table 5.5, it can be observed that different neurons affect the accuracy differently, some have positive effect i.e. breaking these neurons result in an increase of accuracy for that image whereas some have negative effect i.e. breaking these neurons result in loss of accuracy. Also certain times their effect also varies from image to image. For example, while considering the prediction for image 1, it was observed that by breaking one neuron the model can predict the image with better accuracy as compared to when no neurons were broken (i.e. the accuracy prediction increases from 0.652099 to 0.706465) whereas on breaking another neuron, though the model can predict the image correctly but the accuracy by which it predicts drops from 0.652099 to 0.578295 . This trend can be observed in Image 3 as well. However for Image 2, the

trend is violated and accuracy by which it is predicted drops while breaking either of the neuron. The highest accuracy in this case observed was when no neuron is broken. Hence, effect (positive or negative) and importance of neuron vary from image to image.

2. Breaking neurons in Maxpool Layer

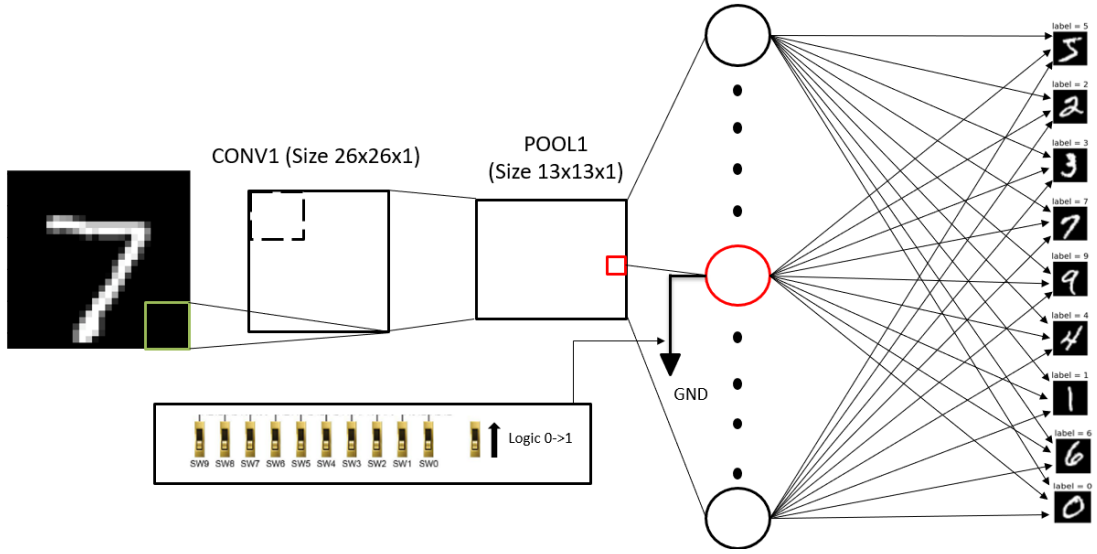

Figure 5.16: Illustration of Breaking Neuron in the Maxpool Layer

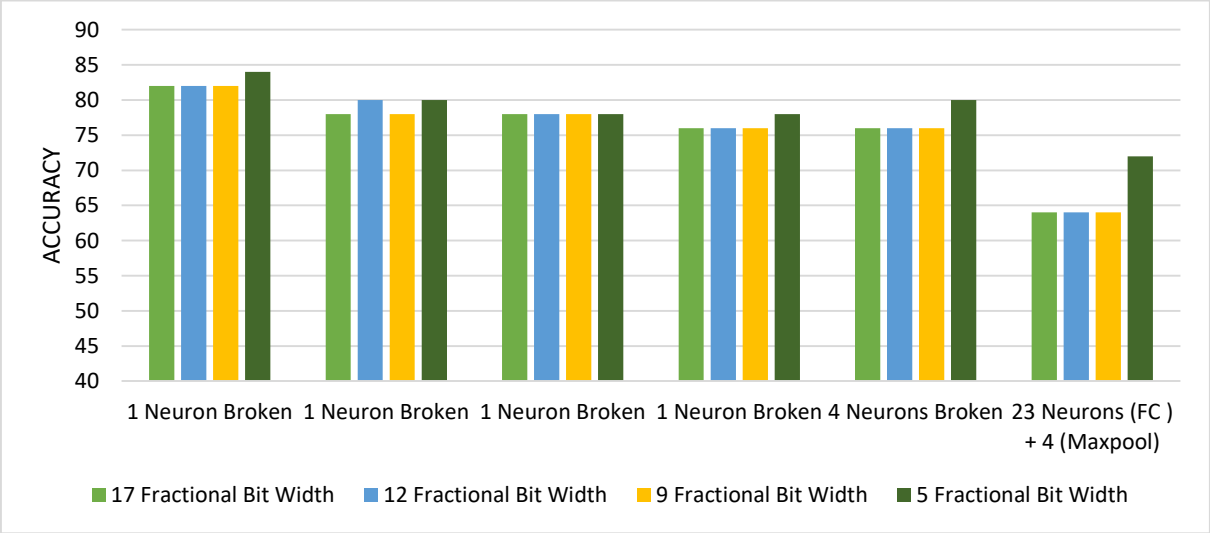

Figure 5.17: Plot comparing obtained accuracy on breaking neurons in all models

Table 5.6: Showing accuracy obtained on breaking neurons in various models

Neurons Broken	**Accuracy**			
	17 Fractional Bit Width	12 Fractional Bit Width	9 Fractional Bit Width	5 Fractional Bit Width
1	82%	82%	82%	84%
1	78%	80%	78%	80%
1	78%	78%	78%	78%
1	76%	76%	76%	78%
4	76%	76%	76%	80%
23 Neurons (FC layer) + 4(Conv Layer)	64%	64%	64%	72%

Figure 5.15 illustrates how the neurons are broken in the max pool layer with the help of FPGA slide switches. Figure 5.16 shows the comparison of accuracy obtained for various models (with various bit width) on breaking different neurons in the max pool layer.

Here, 4 different neurons were broken individually and simultaneously from the architecture. On analyzing the effect of these breaks in neurons on overall accuracy, it was observed that even in the max pool layer the neurons act differently, some have positive effects (increase in accuracy) whereas some have negative effects (drop-in accuracy). For example, in figure 5.16, while breaking one of the neurons, accuracy increases to 82% or 84% whereas on breaking the other neuron accuracy drops to 78% or 76%. Further, on breaking 4 neurons at the same time, the accuracy obtained is still maintained to 76% or 80%. This nature is observed because here some neurons have a negative effect and some have a positive effect for example let's consider the results of 5 Fractional bit Model, as the results show that breaking 1 of the neuron accuracy obtained increase to 84% (positive effect) whereas on breaking other neurons accuracy falls to 78% (negative effect) and on breaking couple of neurons its effect on accuracy is insignificant i.e. no effect on accuracy, When all the 4 neurons are broken are broken simultaneously their effects are

compensated which results into accuracy being maintained to 80%. Hence different neuron effect accuracy differently.

Additionally, it can be observed that with varying bit width effect of neuron on the accuracy also changes. For example, as seen in row 1 of Table 5.5, on breaking one neuron, the accuracy for 17 Fractional bit-width model increases to 82% whereas on breaking the same neuron the accuracy of 5 bit Fractional width model increases to 84%. Also on considering all the factors, it can be observed that the 5 Fractional bit width is comparatively less affected by breaking neurons and utilizes least number of resources.

In the above observation, an additional case is taken where 23 neuron connections are broken from FC to classification layer and 4 neurons are broken in the max pool layer to observe the effect on accuracy when neurons are broken in 2 layers simultaneously. Here, the accuracy gradually falls to 64% for 17, 12, and 9 Fractional bit width models. This is because when neurons are broken individually, their effect on accuracy is insignificant as compared to when they are broken together, it significantly affects the accuracy and accuracy drops by 20%. Whereas for 5 Fractional bit-width model the accuracy achieved is 72%. After breaking the neurons in 2 layers, 5 Fractional bit-width model was retrained to restore the accuracy and two cases were considered here.

Case 1:

The previously trained weights were retrained with very slow learning rate and it was observed that it missed the global minima. The accuracy achieved on retraining the model is 70%.

Case 2:

The weights were trained using initial values (these are the values from which the base model was trained) and the accuracy obtained was 78% which is quite near to the expected accuracy of 80 %.

Hence, the above results conclude that when the accuracy of model drops due to broken neurons, the weights must be retrained from their initial values to restore the accuracy rather than retraining the previously trained weights.

CHAPTER 6

CONCLUSION AND FUTURE WORK

1. Conclusion

The work presents an FPGA implementation of a deep neural network algorithm. This results in a dedicated hardware module for Convolutional Neural Network. Since the model was implemented on hardware, design space exploration was performed to find the optimized model. The results suggest that the 5 Bit Width Model showed an accuracy of 80% with the least resource utilization as compared to other models where accuracy obtained was 80% but resource utilization was significantly higher.

Even though the architecture is fixed, the parameters can be changed which makes the design efficient as even if any connections within the hardware fail, the parameters can be retrained, and depending on the application accuracy can be restored. This gives an upper hand over the other hardware module where failure in connections would make the hardware useless. This increases the efficiency and life expectancy of the hardware and finds its application in self-driving cars.

2. Future Work

There are many aspects in the field of presented work that can be explored further. Some of them are:

- The architecture implemented in the work is shallow compared to the architectures used in many applications. This would increase the size of design and hence design needs to be optimized for minimal resource utilization
- Other learning algorithms can be explored to increase the accuracy
- Application-specific architecture can be developed and re-trained in case some neuron connections break to increase the life expectancy of the hardware module

REFERENCES

[1] A. Ling, J. Anderson, “The Role of FPGA in Deep Learning”, *FPGA '17: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 3, Feb. 2017. https://doi.org/10.1145/3020078.3030013

[2] K, O'Shea, R. Nash, “An Introduction to Convolutional Neural Networks”, 2015. [Online]. Available: arXiv:1511.08458 [cs.NE].

[3] R. Yamashita, M. Nishio, R.K.G. Do, “Convolutional neural networks: an overview and application in radiology”, *Insights Imaging*, vol. 9, no. 4, pp. 611–629, Aug. 2018. https://doi.org/10.1007/s13244-018-0639-9

[4] T. Wang, C. Wang, X. Zhou, H. Chen, “A Survey on FPGA Based Deep Learning Accelerators: Challenges and Opportunities”, 2018. [Online]. Available: arXiv:1901.04988 [cs.DC].

[5] Y. E. Wang, G. Y. Wei, D. Brooks, “Benchmarking TPU, GPU, and CPU Platforms for Deep Learning”, 2019. [Online]. Available: arXiv:1907.10701 [cs.LG]

[6] P. Coussy, D. D. Gajski, M. Meredith and A. Takach, "An Introduction to High-Level Synthesis," in *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 8-17, July-Aug. 2009, doi: 10.1109/MDT.2009.69.

[7] W. Meeus, K. Van Beeck, T. et al. Goedemé, “An overview of today’s high-level synthesis tools”. *Des Autom Embed Syst*, vol. 16, pp. 31–51, Aug. 2012. https://doi.org/10.1007/s10617-012-9096-8

[8] Beginners Guide to Artificial Neural Networks - *https://towardsdatascience.com/from-fiction-to-reality-a-beginners-guide-to-artificial-neural-networks-d0411777571b*

[9] Most popular neural network - *https://www.excella.com/insights/top-3-most-popular-neural-networks*

[10] Structure of Feedforward Neural Network: *https://www.xenonstack.com/blog/artificial-neural-network-applications/*

[11] Explanation of Feedforward Neural Network - *https://towardsdatascience.com/deep-learning-feedforward-neural-network-26a6705dbdc7*

[12] Introduction to RNN - *https://www.analyticsvidhya.com/blog/2017/12/introduction-to-recurrent-neural-networks/*

[13] B. B. Le Cun *et al.* , “Handwritten digit recognition with a back-propagation network”, in *Proceedings of the Advances in Neural Information Processing Systems (NIPS)*, pp. 396–404, 1989.

[14] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, “Gradient-based learning applied to document recognition”, *Proceedings of IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[15] J. Gu *et al.*, "Recent advances in convolutional neural networks", *Pattern Recognition* , vol. 77, issue C, pp. 354-377, May 2018. doi = "https://doi.org/10.1016/j.patcog.2017.10.013".

[16] Role of biases in Neural Networks - *https://www.geeksforgeeks.org/effect-of-bias-in-neural-network/*

[17] Types of Pooling - *https://cs231n.github.io/convolutional-networks/*

[18] Types of Activation functions - *https://medium.com/@abhigoku10/activation-functions-and-its-types-in-artifical-neural-network-14511f3080a8*

[19] Role of regularization and it’s type - *https://towardsdatascience.com/intuitions-on-l1-and-l2-regularisation-235f2db4c261*

[20] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473-491, April 2011. doi: 10.1109/TCAD.2011.2110592.

[21] Y. Guo, D. McCain, J. R. Cavallaro, A. Takach, “Rapid industrial prototyping and SoC design of 3 G/4G wireless systems using an HLS methodology,” *EURASIP J. Embedded Syst.*, vol. 2006, no. 1, pp. 1–25, Jan. 2006

[22] Y. Sun, J. R. Cavallaro, T. Ly, “Scalable and low power LDPC decoder design using high level algorithmic synthesis,” in *Proc. IEEE SoC Conf*, pp. 267–270, Sep. 2009

[23] P. J. Pingree, L. J. Scharenbroich, T. A. Werne, C. M. Hartzell, “Implementing legacy-C algorithms in FPGA co-processors for performance accelerated smart payloads,” in *Proc. IEEE Aerospace Conf.*, pp. 1–8, Mar. 2008

[24] K. Denolf, S. Neuendorffer, and K. Vissers, “Using C-to-gates to program streaming image processing kernels efficiently on FPGAs,” in *Proc. Field Programmable Logic and Application Int. Conf.*, pp. 626–630, 2009

[25] J. Cong and Y. Zou, “Lithographic aerial image simulation with FPGA-based hardware acceleration,” in *Proc. of the 16th Int. ACM/SIGDA symposium on FPGA*, pp. 20–29, Feb. 2008

[26] V. Kindratenko and R. Brunner, “Accelerating cosmological data analysis with FPGAs,” in *17th Proc. of IEEE Field Programmable Custom Computing Machines*, pp. 11–18, 2009

[27] S. Director, A. Parker, D. Siewiorek, and D. Thomas, Jr., “A design methodology and computer aids for digital VLSI,” *IEEE Trans. Circuits Syst.*, vol. 28, no. 7, pp. 634–645, Jul. 1982

[28] A. Parker *et al.*, “The CMU design automation system: An example of automated data path design,” in *Proc. of Design Automation Conf.* , 1979, pp. 73–80

[29] M. R. Barbacci, G. E. Barnes, R. G. G. Cattelland D. P. Siewiorek, “The ISPS computer description language: the symbolic manipulation of computer descriptions”. *Carnegie Mellon University*, June 2018, doi: 10.1184/R1/6610637.v1.

[30] D. D. Gajski, N. D. Dutt, A. C. H. Wu, S. Y. L. Lin, “High-Level Synthesis: Introduction to Chip and System Design”, *Kluwer Academic Publishers*, 1992

[31] D. Wilson, A. Shastri, and G. Stitt, “A High-Level Synthesis Scheduling and Binding Heuristic for FPGA Fault Tolerance”, *International Journal of Reconfigurable Computing*, vol. 2017, pp. 1-17. doi : 10.1155/2017/5419767

[32] S. Davidson, D. Landskov, B. D. Shriver, and P. V. Wallett, "Some Experiments in Local Wicrocode Coapaction for Horizontal Machine". *IEEE Trans. on Computers*. vol. C-30, no.7, pp.460-477, July 1981.

[33] *CyberWorkBench Detail Technical* – White Paper, NEC INDIA PVT LTD., Sep. 2016.

[34] Basic of SystemC - *http://www.asic-world.com/systemc/*

[35] IEEE Standard for Standard SystemC Language Reference Manual," in *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pp.1-638, Jan. 2012, doi: 10.1109/IEEESTD.2012.6134619

[36] I. Kuon, R. Tessier, and J. Rose, "FPGA Architecture: Survey and Challenges," in *FPGA Architecture: Survey and Challenges*, 2008

[37] FPGA Architecture Description - *https://www.elprocus.com/fpga-architecture-and-applications/*

[38] Structure of Logic Block - *https://en.wikibooks.org/wiki/Programmable_Logic/FPGAs*

[39] Advantage of ReLU - *https://towardsdatascience.com/complete-guide-of-activation-functions-34076e95d044*

[40] *Xilinx Vivado Design Suite High Level Synthesis* – UG902, v2019.2 ed., Xilinx, Jan. 2020.

[41] Design flow of CyberWorkBench tool - *https://www.nec.com/en/global/prod/cwb/index.html*

[42] *FPGA Architecture* – White Paper, ver. 1.0, Altera, July 2006.

[43] *Terasic DE1 SoC User Manual* – Terasic Technologies Inc. URL: *www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=205&No=836&PartNo=4*

[44] X. Li, G. Zhang, H. H. Huang, Z. Wang, W. Zheng, "Performance Analysis of GPU-Based Convolutional Neural Networks," *2016 45th International Conference on Parallel Processing (ICPP)*, pp. 67-76, 2016. doi: 10.1109/ICPP.2016.15.

[45] MNIST dataset description - *http://yann.lecun.com/exdb/mnist/*

[46] Introduction to Gradient Descent - *https://towardsdatascience.com/understanding-the-mathematics-behind-gradient-descent-dde5dc9be06e*

[47] Plot between weight and cost function - *https://blog.clairvoyantsoft.com/the-ascent-of-gradient-descent-23356390836f*

[48] Effect of Learning Rate on Cost Function - *https://www.educative.io/edpresso/learning-rate-in-machine-learning*

[49] What is overfitting and what are it's effect? - *https://towardsdatascience.com/what-are-overfitting-and-underfitting-in-machine-learning-a96b30864690*

[50] DE1-SoC My First HPS-FPGA - Terasic Technologies Inc. URL: *https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&No=836&PartNo=4*

BIOGRAPHICAL SKETCH

Akshay Raju Krishnani was born in Veraval, in the state of Gujarat in India on August 5, 1996 as the eldest son to Raju and Geeta Krishnani. He earned his bachelor's degree in Electrical Engineering from Nirma University in 2018. His research work on the Gesture-Based Wireless system led to a major publication in the International Journal of Research in Engineering and Technology. Apart from academics, his acumen in marketing skills led him to serve as a Marketing Head of Rotaract Club of Nirma Institutes.

Later on, Akshay pursued a master’s degree program in Electrical Engineering with a specialization in computing systems at The University of Texas at Dallas. He worked at Intel Corporation as a Logic Design Engineer intern with the Media Graphics Pre-Silicon Validation team. He has been a member of the DARClab (Design Automation and Reconfigurable Computing lab) since January 2019. His research interests includes hardware accelerators, reconfigurable computing, AI-based algorithms, and design automation tools.

CURRICULUM VITAE

Akshay Raju Krishnani

July 16, 2020

Contact Information:

Department of Electrical Engineering

The University of Texas at Dallas

800 W. Campbell Rd.

Richardson, TX 75080-3021, U.S.A.

Email: akshay.krishnani05@gmail.com

Educational History:

M.S.E.E, The University of Texas at Dallas, 2018-Present

MSEE Thesis: In-situ Implementation and Training of Convolutional Neural Network

Thesis Advisor: Dr. Benjamin Carrion-Schaefer

B.Tech, Electrical Engineering, Institute of Technology Nirma University, India, 2018

Work Experience:

Logic Design Engineer Intern, Intel Corporation, Folsom, CA (August 2019-January 2020)