



# Deployment of Deep Neural Networks on FPGAs

## Final Year Project Report

**by**  
Abdullah Ashfaq  
&  
Syed Tihaam Ahmad

**Advisor**  
Dr. Faisal Shafait

**Co-advisor**  
Dr. Muhammad Shahzad

In Partial Fulfillment of the Requirements for the degree  
Bachelor of Electrical Engineering (BEE)

School of Electrical Engineering and Computer Science  
National University of Sciences and Technology Islamabad, Pakistan  
(2020)

## Declaration

We hereby declare that this project report entitled "Deployment of Deep Neural Networks on FPGAs" submitted to the "Department of Electrical Engineering, SEECS", is a record of an original work done by us under the guidance of Supervisor "Prof. Dr. Faisal Shafait" and that no part has been plagiarized without citations. Also, this project work is submitted in the partial fulfillment of the requirements for the degree of Bachelor of Electrical Engineering.

Team Members	Registration No.
Abdullah Ashfaq	129399
Syed Tihaam Ahmad	177607

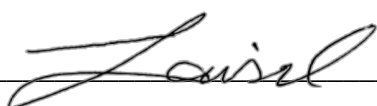
Date: 15<sup>th</sup> June, 2020

## Approval

It is certified that the contents and form of thesis entitled “Deployment of Deep Neural Networks on FPGAs” submitted by Abdullah Ashfaq (Reg. 00000129399) & Syed Tihaam Ahmad (Reg. 00000177607) have been found satisfactory for the requirement of the degree.

**Advisor:**

Dr. Faisal Shafait



---

**Co-Advisor:**

Dr. Muhammad Shahzad

---

## **Dedication**

We dedicate this project to Allah Almighty, our supervisors and our parents, whose constant support, motivation and belief in our abilities led us to wherever we are right now.

Abdullah Ashfaq  
&  
Syed Tihaam Ahmad

## Acknowledgments

We are highly grateful to our thesis advisors, Dr. Faisal Shafait and Dr. Muhammad Shahzad, for their guidance, support and supervision throughout the project.

We would also like to thank Mr. Ussama Zahid, Research Intern at Xilinx, and Mr. Muhammad Umar, PhD student Cornell, for providing us their implementations and work in the form of report and guiding us whenever we asked for help.

We would also like to thank Mr. Muhammad Mohsin Ghaffar, M.Sc., of the Microelectronic Systems Design Group, TU Kaiserslautern, for his valuable insights and feedback.

We would like to acknowledge the open-source efforts of Xilinx Corporation, especially in releasing their pioneering work on FPGA based Deep Learning IPs, which we used as a platform for our project.

Finally, we would like to thank TU Kaiserslautern and DAAD Germany for funding the programmable logic hardware used in our project.

## Table of Contents

Declaration .....	1
Approval .....	2
Dedication .....	3
Acknowledgments.....	4
Table of Contents .....	5
List of Tables .....	8
List of Figures .....	9
List of Abbreviations .....	12
Abstract .....	1
<b>Chapter 1 Introduction.....</b>	<b>2</b>
1.1 Deep Neural Networks .....	2
1.1.1 Fully Connected (FC) Layer .....	3
1.1.2 Batch Normalization Layer.....	3
1.2 Convolutional neural Network .....	4
1.2.1 Convolutional Layer .....	5
1.2.2 Pooling layer .....	6
1.3 FPGAs .....	7
1.3.1 System on Chip (SoC) FPGAs.....	7
1.3.2 Vivado Toolchain.....	7
1.3.3 FPGA board .....	10
<b>Chapter 2 Literature Review .....</b>	<b>12</b>
2.1 Binary Neural Networks (BNN) .....	12
2.2 Quantized Neural Networks (QNN).....	12
2.3 Winograd Convolution.....	13
2.4 Data Tiling.....	15

2.5	Classification of Hardware Accelerators: .....	15
2.6	Why Int8?.....	16
2.7	Post Training Quantization.....	17
2.7.1	Fused Layers .....	17
2.7.2	Using Probability .....	18
<b>Chapter 3</b>	<b>Problem.....</b>	<b>19</b>
3.1	Shortcomings in Previous work .....	19
3.2	Methodology .....	19
3.3	Our Objectives.....	20
3.4	Workflow .....	20
<b>Chapter 4</b>	<b>Design and Implementation .....</b>	<b>22</b>
4.1	Model Training.....	22
4.1.1	BREVITAS - By Xilinx.....	22
4.1.2	Batch Norm layer .....	22
4.1.3	Networks Tested .....	23
4.2	Software Simulation.....	24
4.2.1	Weight Packing.....	26
4.3	Hardware Design.....	26
4.3.1	Stage 1: C++ design with floating point operations .....	26
4.3.2	Stage 2: Non-dataflow hardware design with fixed point operations.....	26
4.3.3	Final Stage: Dataflow design with integer operations .....	27
4.3.4	Vivado Cosimulation .....	31
4.4	Hardware Synthesis and Implementation.....	33
4.5	Running Models on Hardware .....	35
4.5.1	Xilinx SDK .....	35
4.5.2	PYNQ Linux .....	36
<b>Chapter 5</b>	<b>Challenges.....</b>	<b>38</b>

5.1	Training .....	38
5.2	Challenges with Winograd .....	38
5.2.1	Precision .....	38
5.2.2	Memory .....	39
5.3	Over-utilization of Resources .....	39
5.4	Control Sets .....	39
5.5	Hardware Debugging .....	40
<b>Chapter 6</b>	<b>Results and Discussion .....</b>	<b>41</b>
6.1	Lenet – MNIST .....	41
6.2	Cifar10- CNV .....	43
6.3	CHaiDNN .....	44
<b>Chapter 7</b>	<b>Conclusion .....</b>	<b>45</b>
7.1	Future Work .....	45
<b>Appendix .....</b>		<b>46</b>
i)	Github Repositories .....	46
ii)	Sample PYNQ driver .....	47
iii)	Sample HLS top function .....	48
iv)	Sample SDK function .....	48
<b>Bibliography and Webography .....</b>		<b>51</b>



## List of Tables

	<b>Pg</b>
Table 1 - PYNQ-Z1 Resources .....	11
Table 2 - Accuracy for Quantized Neural Networks .....	18
Table 3 - Lenet-MNIST Structure.....	23
Table 4 - CIFAR10-CNV Structure.....	24
Table 5 - Unoptimized Utilization .....	27
Table 6 - Lenet-MNIST Hardware Design Comparison.....	41
Table 7 - Lenet-MNIST on different Platforms .....	43
Table 8 - CIFAR10-CNV Hardware Design Utilization .....	43
Table 9 - CIFAR10-CNV on Different Platforms .....	44
Table 10 - CHaiDNN Results on ZC706 .....	44

## List of Figures

	<b>Pg</b>
Figure 1 - Basic Neuron.....	2
Figure 2 - Deep Neural Network with Multiple Layers.....	3
Figure 3 - Fully Connected Layer.....	3
Figure 4 - Batch Norm Layer.....	4
Figure 5 - Convolutional Neural Network.....	4
Figure 6 - CNN for MNIST.....	5
Figure 7 - Convolution operation.....	5
Figure 8 - Maxpool Layers.....	6
Figure 9 - FPGA SoC.....	7
Figure 10 - HLS Design Flow.....	8
Figure 11 - Vivado IP Integrator.....	9
Figure 12 - Xilinx SDK.....	10
Figure 13 - PYNQ-Z1.....	10
Figure 14 - Zynq 7000 Block Diagram.....	11
Figure 15 - Dataflow Architecture.....	12
Figure 16 - Multi-layer offload.....	13
Figure 17 - Normal Conv2d.....	13
Figure 18 - Input transformation.....	14
Figure 19 - Kernel transformation.....	14

Figure 20 - Output transformation .....	14
Figure 21 - Tiling .....	15
Figure 22 - Dataflows for DNN .....	16
Figure 23 - Repeating Block in CNN .....	18
Figure 24 - Efficient ways of DNN Inference .....	20
Figure 25 - Workflow .....	21
Figure 26 - Lenet-MNIST diagram.....	23
Figure 27 - CIFAR10-CNV diagram .....	24
Figure 28 - Software Simulation.....	25
Figure 29 - Instruction Level Parallelism .....	28
Figure 30 - Different Types of Hardware Designs .....	28
Figure 31 - Specifications of Different Types of Hardware Designs .....	28
Figure 32 - Basic Compute Engine.....	29
Figure 33 - Compute Engine with 2 SIMD lanes .....	30
Figure 34 - Compute Engine with 2 SIMD lanes and computes 4 outputs in parallel .....	30
Figure 35 - Input Sharing [7] .....	31
Figure 36 - DSP Slice with MACC mode [7] .....	31
Figure 37 - RTL Co-simulation output .....	32
Figure 38 - Co-simulation waveforms .....	32
Figure 39 - Deep Neural Network IP.....	33
Figure 40 - System Block Design .....	33

Figure 41 - Implementation Summary .....	34
Figure 42 - Implementation Result .....	34
Figure 43 - Xilinx SDK Result .....	36
Figure 44 - PYNQ Notebook Sample .....	37
Figure 45 - Lenet-MNIST Design Space Exploration .....	42
Figure 46 - Summary of Power Utilization of SoC .....	42
Figure 47 - CIFAR10-CNV Hardware Design .....	43
Figure 48 - Github Repository .....	46
Figure 49 - PYNQ Driver .....	47

## List of Abbreviations

CNN – Convolutional Neural Networks

DNN – Deep Neural Networks

BNN – Binarized Neural Networks

FPGA – Field Programmable Gate Arrays

PE – Processing Elements

SIMD – Single Instruction, Multiple Data

PL – Programmable Logic

PS – Processing System

## **Abstract**

Deep learning is applied in modern technologies from self driving cars to detection drones. For it to be practical in real time, DL models are implemented on hardware. GPUs and CPUs are main devices which handle billions of mac operations real time. But they have a major drawback of latency and excessive power consumption. So in order to cater this problem we deployed the same deep neural network models on Zynq FPGAs to achieve similar results but with high throughput and minimum power consumption. This made it practical to deploy deep learning networks on real-time devices.

---

# Chapter 1 Introduction

In this chapter we will be over-viewing basics of DL and Xilinx tools just to refresh the knowledge of the reader.

## 1.1 Deep Neural Networks

DL is considered a subset of machine learning. Neural networks are the core component of deep learning. Neural network was inspired by the working of the brain. An artificial neuron is fed with an input, which then gives an amplified or biased value in the output. A simple artificial neural network can be represented as:

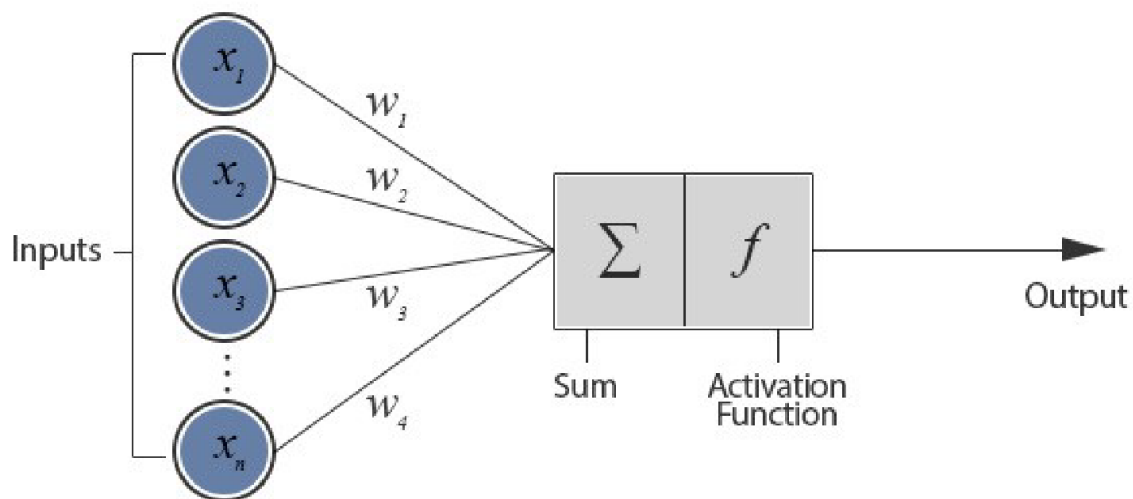


Figure 1 - Basic Neuron

Deep neural networks have layers of such and similar artificial neural layers. Such networks are fed with labelled inputs and the weights are tuned to sweet spots such that the network can predict inputs other than the ones it was trained on called test inputs. The first and last layers are usually referred to as input and output layers, respectively. And the neural layers in-between are called hidden layers. One such deep neural network is shown below.

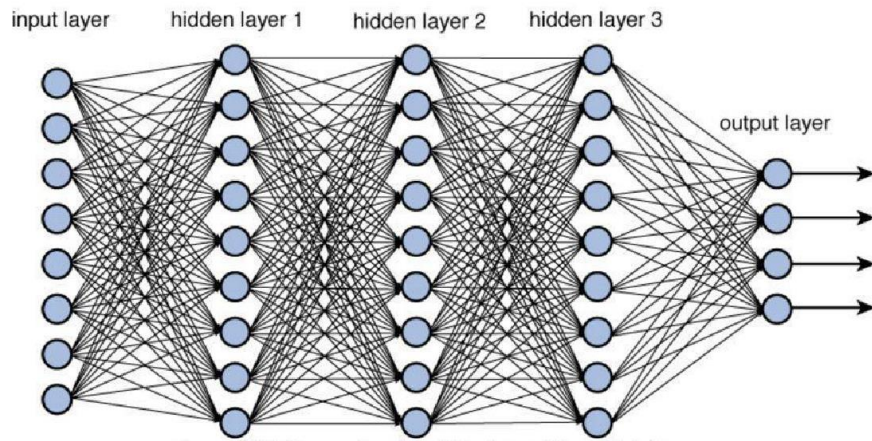


Figure 2 - Deep Neural Network with Multiple Layers

These neural layers can be of different types:

### 1.1.1 Fully Connected (FC) Layer

Fully connected layer also known as densely connected layer is a very common layer used in deep neural networks. In this layer, every neuron receives all the inputs from the previous layer and gives a single output. This layer will be implemented in our Winograd based library and can be used on PL and PS sides depending on the nature of architecture.

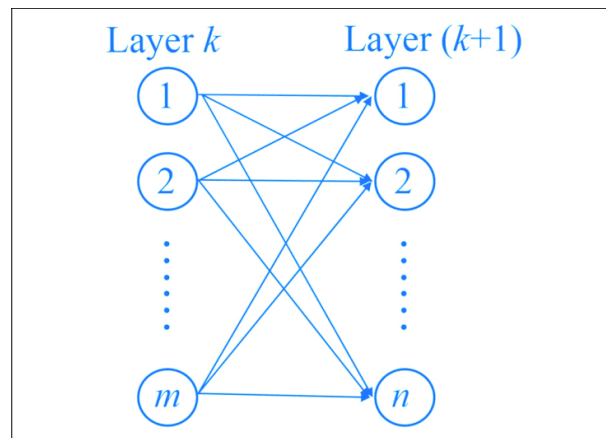


Figure 3 - Fully Connected Layer

### 1.1.2 Batch Normalization Layer

A batch norm layer is usually used to make the training process faster by normalizing the values according to mean and variance values learned along the process of training weights.



This layer is usually not required in the inference process.

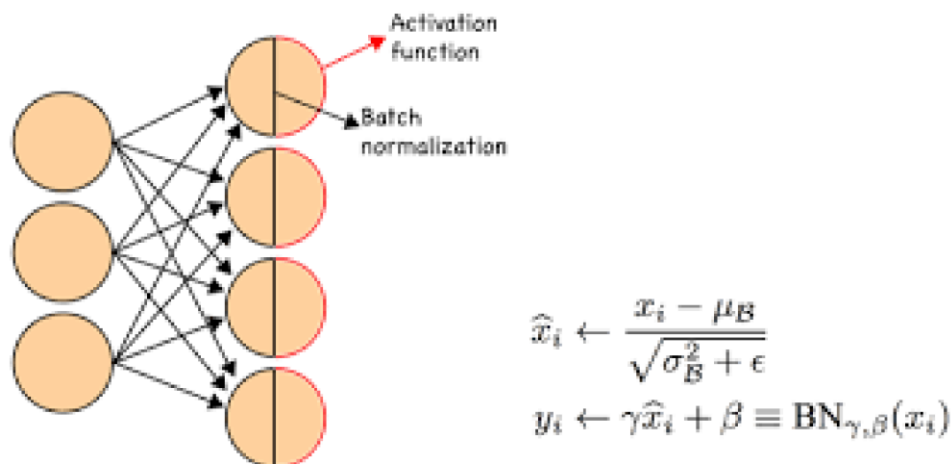


Figure 4 - Batch Norm Layer

## 1.2 Convolutional neural Network

CNNs are deep neural networks used as black boxes which are fed with labelled inputs and they train themselves by automatic feature extraction. The old techniques of using hand-crafted features for classifying images are tedious and time consuming. Hence, these architectures are the state-of-the-art networks for visual recognition and object classification.

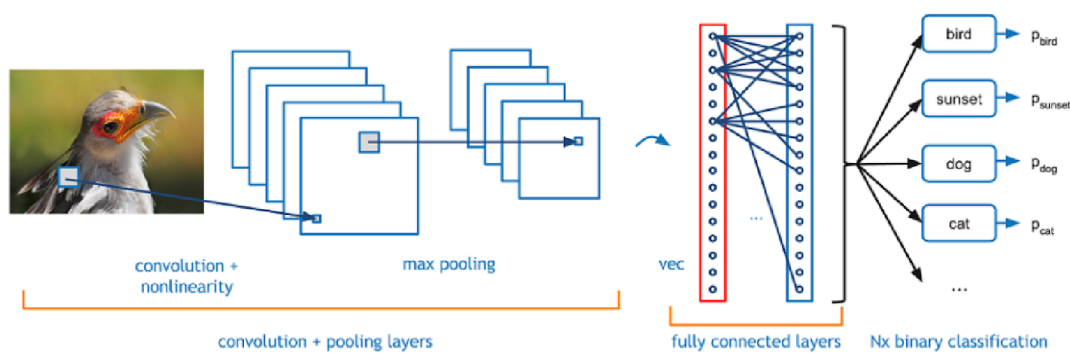


Figure 5 - Convolutional Neural Network

A CNN is made up of different convolutional, pooling and fully connected layers. A simple LeNet used for handwritten digits' classification is shown below:

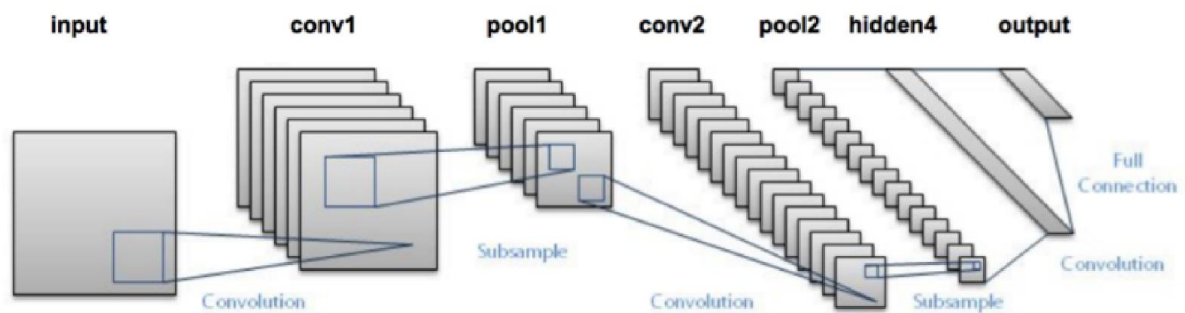


Figure 6 - CNN for MNIST

### 1.2.1 Convolutional Layer

A convolutional layer performs a simple conv 2d on the input image. Every conv layer has a set of filters that perform convolution on the input fed to the layer. This filter bank has multiple filters and each filter is convoluted with different layers of fed input.

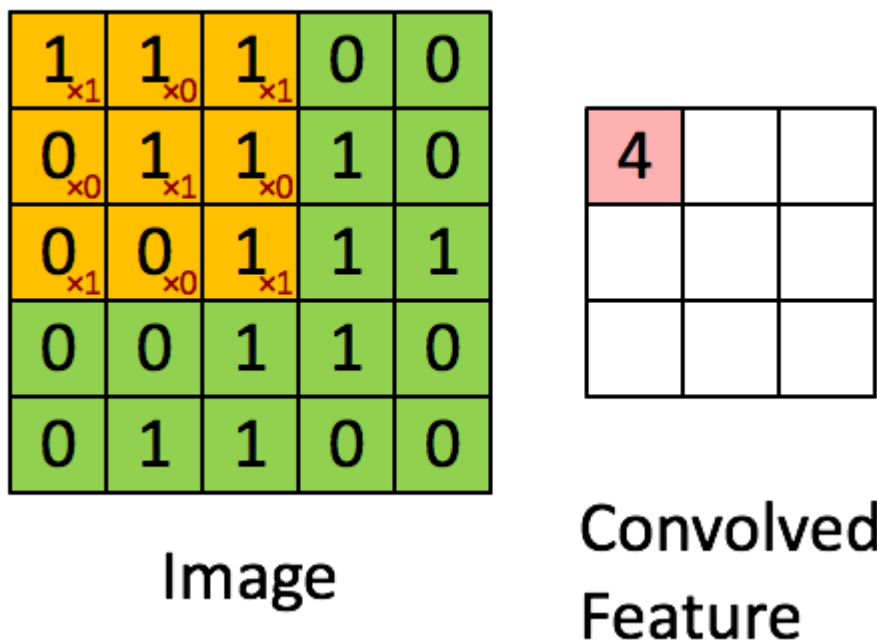


Figure 7 - Convolution operation

We will be implementing a special type of convolution i.e Winograd convolution in our library which is also a novelty in our work. We will be discussing Winograd convolution in detail in the coming sections.

### 1.2.2 Pooling layer

A pooling layer is used to subsample and lower the dimension of the feature map. It is used to discard the redundant information in the feature map and pass the useful information to the following layers. Two types of pooling layers are commonly used which are perfectly depicted in this image:

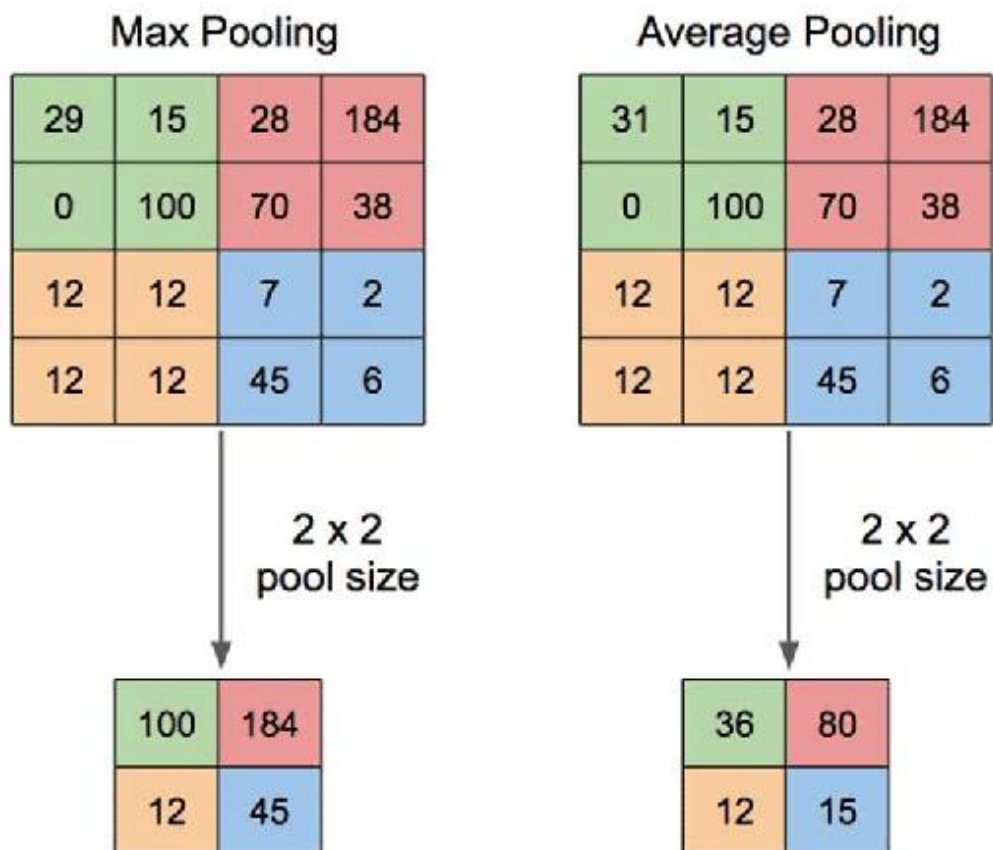


Figure 8 - Maxpool Layers

## 1.3 FPGAs

Field Programmable Gate Arrays are reconfigurable chips used to implement integrated circuits for specified applications. A FPGA is actually based on Lookup tables LUTs, Configurable logic blocks for logic implementation, BRAM ,SRAM cells for memory purposes and I/O blocks for interfacing and communication. Any custom hardware can be implemented using its functionalities considering the resource constraints.

### 1.3.1 System on Chip (SoC) FPGAs

SoC FPGAs consist of two main parts PS and PL. Processing system PS is basically an ARM processor chip and Programmable logic PL consist of FPGA chip. Many companies like Intel and Xilinx manufacture and sell such SoCs. We will be using two of such SoC FPGA boards called PYNQ and Zynq 706.

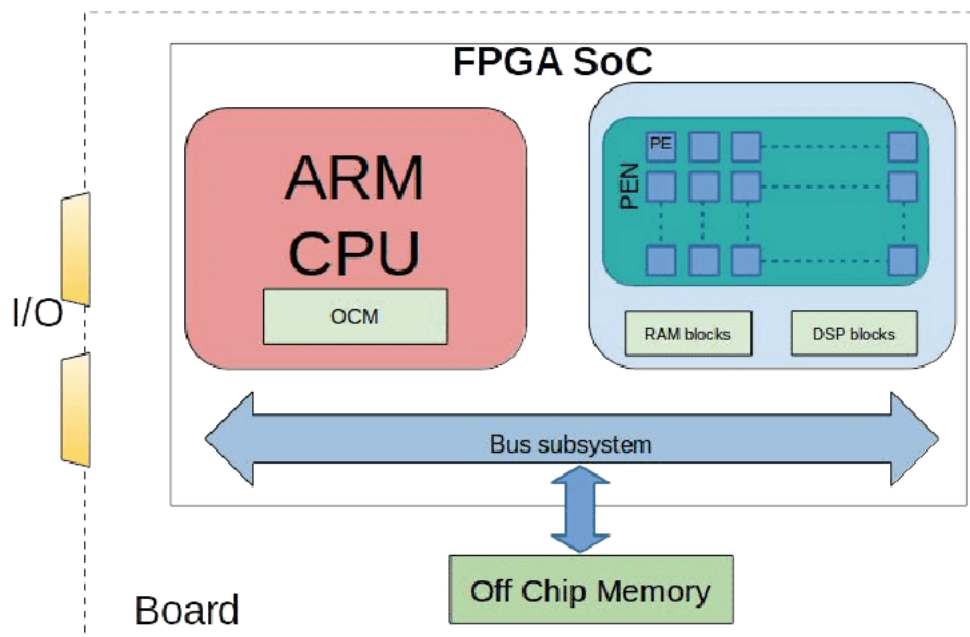


Figure 9 - FPGA SoC

### 1.3.2 Vivado Toolchain

Xilinx Vivado toolchain is used in order to work with these boards. Vivado 2018.2 was mainly used in our FYP for implementing the HLS based custom library on the FPGA boards. Vivado

HLS was used to write the library and run the simulations but in order to run it on the board we used Vivado SDK.

### 1.3.2.1 Vivado HLS

Vivado HLS converts a C/C++ like code into RTL code for synthesis and simulation. High level synthesis (HLS) was used to write the our library and simulations were obtained on Vivado HLS. It provides an abstraction to write the hardware code. It includes specific pragmas defining the hardware structure and style along with basic C/C++ coding. A custom IP is finally generated which can then be used in interfacing on

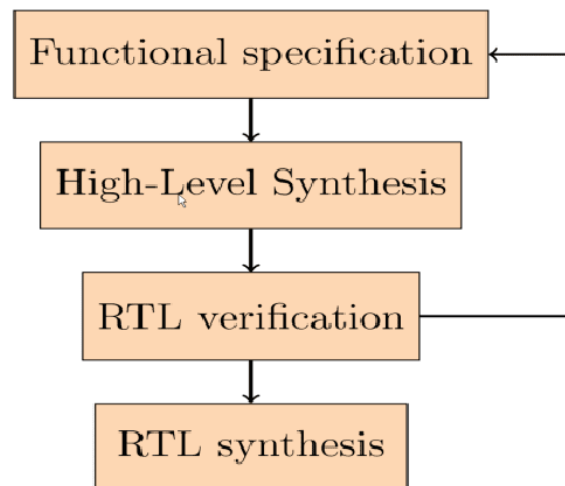


Figure 10 - HLS Design Flow

### 1.3.2.2 Vivado IP Integrator

After getting our custom IP from Vivado HLS, Vivado IP integrator is used to map this custom IP on our FPGA board. Xilinx offers a GUI based integrator in order to speed up and ease the process. This IP integrator is used to connect different IP blocks and interface them with PS if we want to. It provides timing analysis reports ,power estimates and resource utilization reports. Finally it provides the bitstream which is uploaded to the FPGA board in order to program it with the custom design.

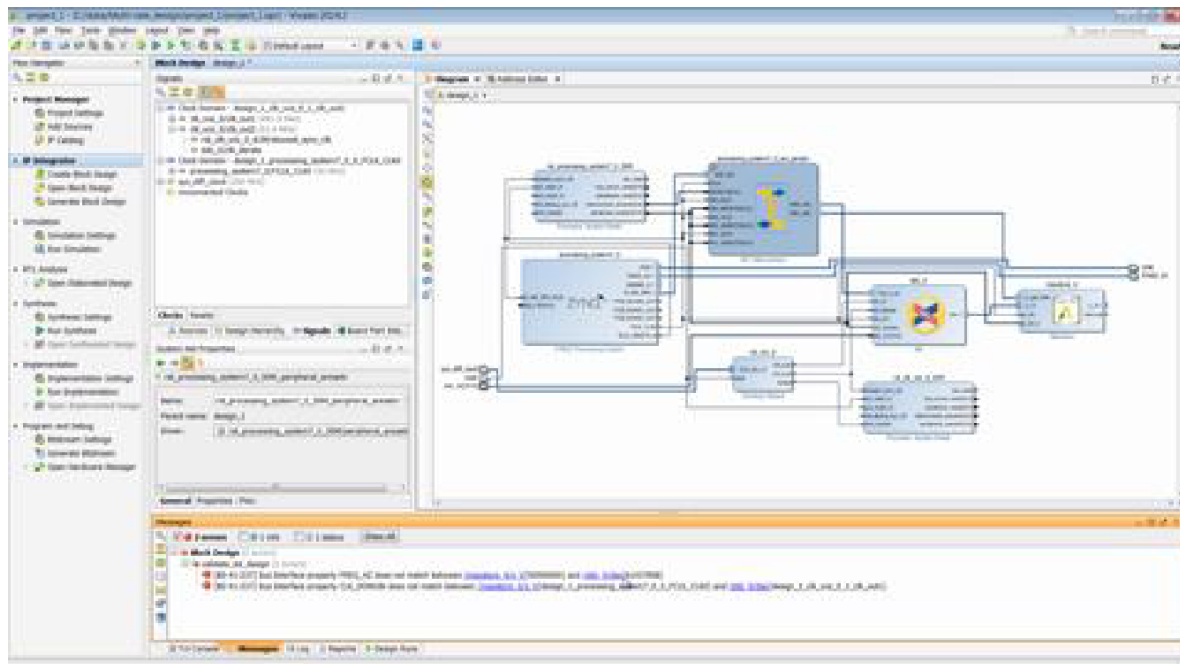


Figure 11 - Vivado IP Integrator

### 1.3.2.3 Xilinx SDK

Xilinx SDK is a software development kit used to program the PS part for the Zynq device. A code is written and implemented in order to run on ARM processor which controls and communicates with the PL part.



Resource	Available
LUT	53,200
LUTRAM	17,400
FF	106,400
BRAM	140
DSP	220

Table 1 - PYNQ-Z1 Resources

The block diagram of Zynq-7000 series boards is as follows [1, p. 70].

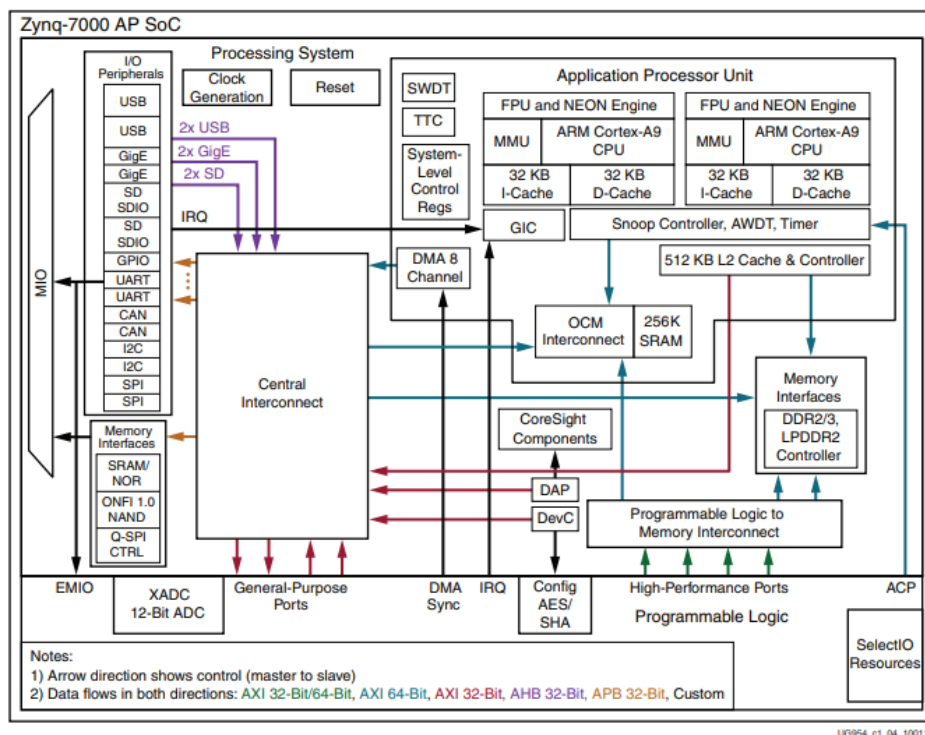


Figure 14 - Zynq 7000 Block Diagram

The thing to note is that there are two AXI 64-Bit connections between programmable logic and DDR memory. To utilize this the maximum bandwidth, we used a custom DMA to transfer 64-Bits at a time. 64-Bits are used for input and 64-Bits are used for writing outputs back to DDR.



## Chapter 2 Literature Review

### 2.1 Binary Neural Networks (BNN)

Low bit width weights and activations were used in hardware in order to remove redundancy and calculation off the hardware. Binary neural networks were introduced in this regard [2]. In binary neural network the weights and activations are binarized to  $+1/-1$ . BNNs made a significant position in the hardware world from a deployment point-of-view because of its advantages in terms of computations. When activations and weights are both binary, the usual mac operation can be replaced by an XNOR-popcount. The usual multiply-accumulate consists of a dot product between two vectors and adding the resulting numbers. This is replaced by bitwise XNOR of the two vectors. The core BNN architecture is a dataflow architecture and is shown as below :

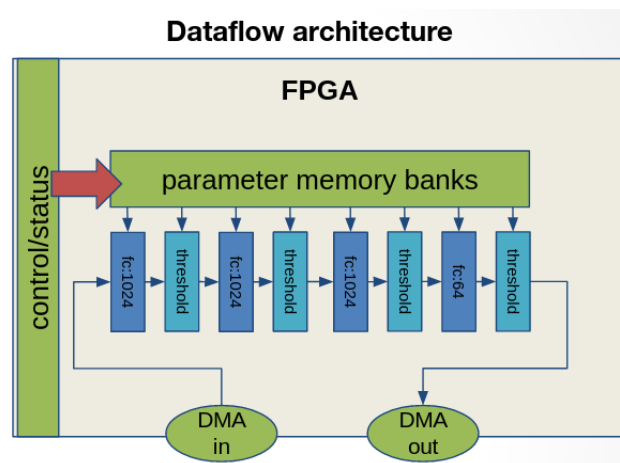


Figure 15 - Dataflow Architecture

### 2.2 Quantized Neural Networks (QNN)

QNN are also general low bit width networks but the weights and activations in QNN are not constrained to 1-bit but can have an arbitrary number of bits. Sometimes for bigger networks the binarized networks have way too much accuracy trade off with lower number of bits hence quantized neural networks are used. It uses multi-layer offload architecture shown as below :

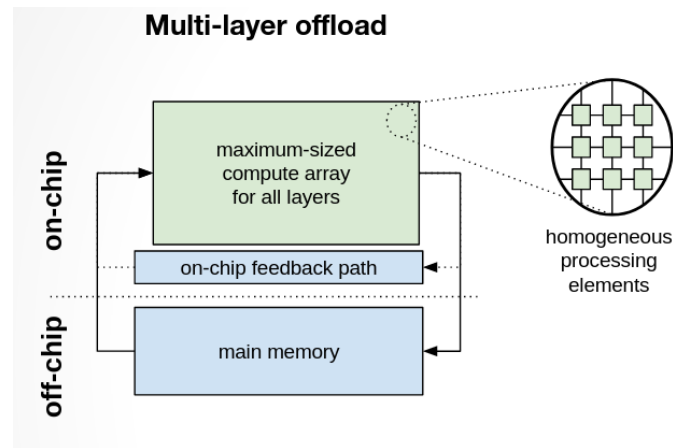


Figure 16 - Multi-layer offload

## 2.3 Winograd Convolution

Convolution operation is considered the backbone of a convolutional neural network. A normal 2d convolution uses 36 mac operations for 3x3 filter over a 4x4 image. It maps out to billions of operations for a big DL network.

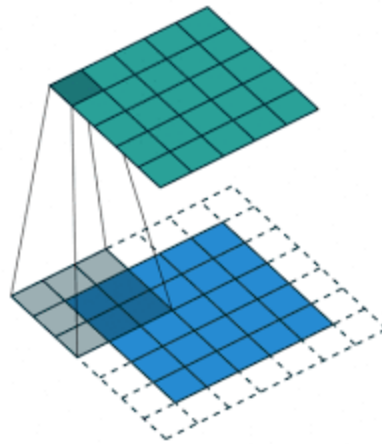


Figure 17 - Normal Conv2d

On the other hand, Winograd convolution [3] [4] for the same parameters takes about 16 mac operations. Hence, it can optimize the convolution operation by 2.25x. To achieve this performance, we have to perform Winograd transformations on input and weights. Pre-trained weights are already transformed while inferring an input. Hence, it does not affect the real-time throughput of the accelerated hardware. There are some problems of accuracy with Winograd

convolution which are being addressed [5]. The following images depict the Winograd special matrices used for transformation in case of (4x4 image, 3x3 filter, Stride 1) :

$$\begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix} = \left( \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix} \right) \cdot \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}^T$$

Figure 18 - Input transformation

$$\begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix} = \left( \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \\ * & * & * \end{bmatrix} \right) \cdot \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}^T$$

Figure 19 - Kernel transformation

Element wise product results in usage of less MACs than normal convolution (2.25x less MACs).

$$\begin{bmatrix} * & * \\ * & * \end{bmatrix} = \left( \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \cdot \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix} \right) \cdot \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}^T$$

Figure 20 - Output transformation

In order to extend this 4x4 convolution to our bigger inputs such as (28x28 MNIST, 32x32x3 CIFAR etc) we have to tile the data 4x4 form and then stitch it every time after obtaining outputs. But this operation is usually done on the PS side.

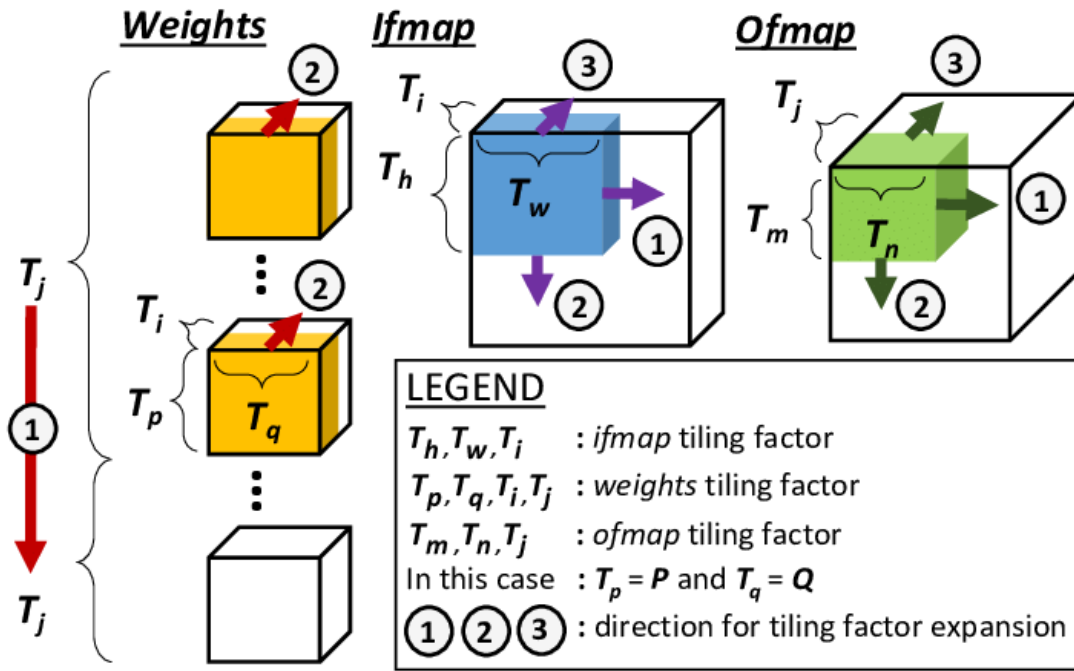


Figure 21 - Tiling

## 2.4 Data Tiling

When implementing our complete Winograd-based architecture [6] on a FPGA board we had two choices either to go with multi-layer offload or dataflow architecture. Dataflow architecture is a one-go architecture and it does not need to offload the architecture after every layer. That is why it's more efficient in terms of throughput. Both architectures can easily be understood by looking into the following images.

## 2.5 Classification of Hardware Accelerators:

The deep neural network accelerators can be classified into following categories:

### 1. Weight Stationary

In order to minimize the movement of weights from DRAM, these can be locally stored in register files. This allows filter reuse. The inputs are moved through the PEs but the weights remain stationary.

### 2. Output Stationary

To get an output of a filter, there are a number of partial sums which need to be accumulated. These are kept in a register file and weights are broadcasted to PEs. There are a number of implementations of this category, one of which is shown in image below.

### 3. No Local reuse

Although register files allow energy to be saved by caching values close to processing engine, they are inefficient in term of area. Nothing remains stationary in the PE and all the space is provided to the global buffer. As a result, there is more traffic in this implementation as compared to other categories.

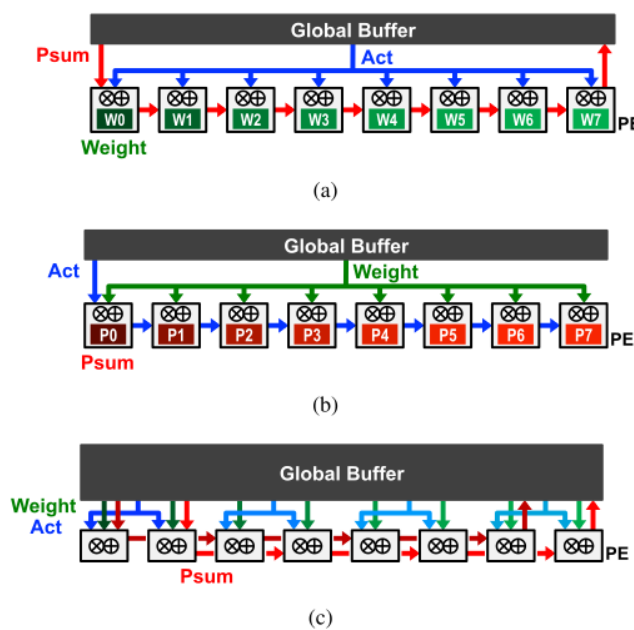


Figure 22 - Dataflows for DNN

[7] a)Weight Stationary b)Output stationary c)No local reuse

## 2.6 Why Int8?

According to a white paper by Xilinx “Xilinx INT8 optimization provides the best performance and most power efficient computational techniques for deep learning inference. Xilinx's integrated DSP architecture can achieve 1.75X solution-level performance at INT8 deep learning operations than other FPGA DSP architectures”. [8]

From the software point of view, it has already been proved that deep learning models have enough degree of freedom to achieve the same results with int8 parameters as with full

---

precision parameters. But int8 parameters provide an advantage in the implementation. Instead of using resource expensive floating-point units, we can use cheaper integer units for computation. Integer units are also faster than the floating-point units.

Due to reason such as these, we opted to use int8 bitwidth for the parameters of our models.

There are two ways in which this can be achieved:

1. Quantization-aware training
2. Post training quantization

Our hardware design is capable of running models from both these techniques.

## **2.7 Post Training Quantization**

By post-training quantization, we mean that we do not have access to complete dataset. As a result, we do not have the luxury of fine-tuning to regain the accuracy lost as a result of quantization. A number of approaches have been studied in this regard [9]. This demands for using novel techniques to prevent loss of accuracy. Some techniques which we implemented are as follows:

1. Fused Layers
2. Using Probability

### **2.7.1 Fused Layers**

There is a pattern of Convolution layer followed by batch normalization and scaling layer in many neural networks including Resnet and Googlenet. These are linear operations and hence can be fused into a single convolution layer with weights and biases. In this way, there are less parameters to quantize and hence accuracy drop is minimum.

---

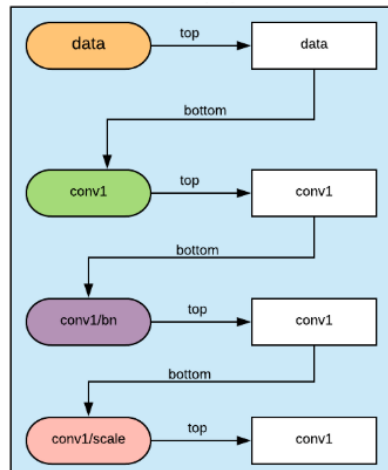


Figure 23 - Repeating Block in CNN

If this combination is followed by ReLU activation, it can be fused with this block to further increase resolution by 1 bit because ReLU sets negative activations to zero.

### 2.7.2 Using Probability

In many cases, it can be observed that weights follow a bell curve. So we can assume that they follow a normal distribution. As we know that most of the density for normal distribution falls within  $3\sigma$  we can use that value to scale the weights instead of the absolute maximum value as previously done.

Our results for 16-bit and 8-bit quantization using the above two methods are mentioned in table below. We tested our results on 100 images of Imagenet dataset.

Network	32-bit	16-bit	8-bit
Mobilenet	69%	69%	69%
Googlenet	67%	66%	66%
Resnet	75%	75%	74%

Table 2 - Accuracy for Quantized Neural Networks

---

## Chapter 3 Problem

GPUs are easy to use and mostly used in DNN inference but they come with a price. They are power hungry and give less cycles/watts generally. Hence, to accelerate power consuming DNN slow inference problems in critical scenarios we will be using low latency, super fast and low powered FPGAs. FPGAs will be used to meet domain-specific time, accuracy and power constraints. Hence, increasing cycles/watts in inference of deep learning models.

### 3.1 Shortcomings in Previous work

There has been work in this domain for some time now. The most notable is the BNN and the QNN repositories of Xilinx. We explored Xilinx's BNN and QNN and deduced the following shortcomings:

- Time-consuming training
- Failure of BNN when deployed for larger models
- Loss of accuracy
- QNN decreases FPS

There seems to be an opportunity in the algorithm for convolution. Different algorithms such as Winograd and FFT based implementations have not been vastly experimented with yet. We decided to target these different algorithms.

### 3.2 Methodology

We needed a platform that is

- Fast
  - Low powered
  - Customizable
-



We decided to use FPGAs as these qualify all these categories. From the software side, we decided to use quantization to reduce size of parameters, as well as reduce the footprint of the hardware design.

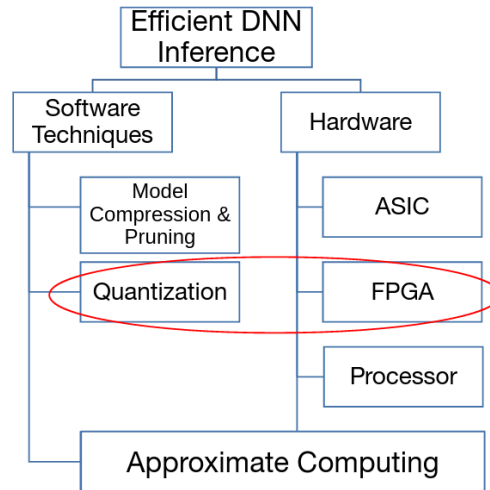


Figure 24 - Efficient ways of DNN Inference

### 3.3 Our Objectives

We aimed to provide generic and versatile building blocks in HLS

- For variety of networks
- User-friendly experience when deploying a model
- Complete workflow from training to inference
- Target Xilinx Zynq FPGAs
- Quick exploration of design space

### 3.4 Workflow

The workflow that we took started with model training. Then we quantized the parameters and translated then into Winograd domain to use the more efficient Winograd algorithm during inference.

We designed the hardware for Winograd convolution to require less computations than a standard deep neural network accelerator.

Our end product will be an IP which can be seamlessly integrated into a design and controlled through the PS.

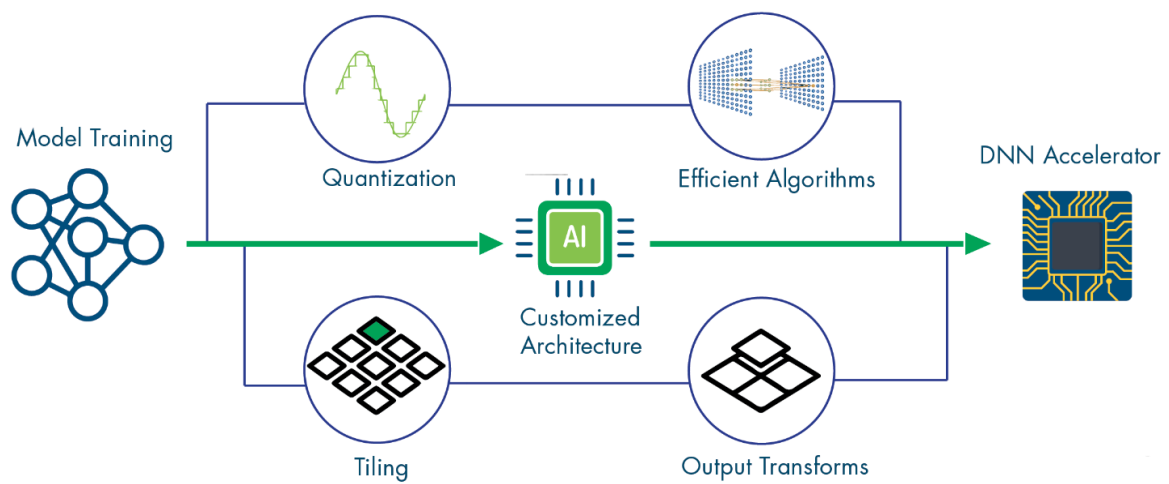


Figure 25 - Workflow

---

## Chapter 4 Design and Implementation

As we were aiming at an end-to-end complete pipeline to deploy deep learning models on FPGAs, our process went through a number of stages which can be categorized as follows:

1. Model Training
2. Software Simulation
3. Hardware Design
4. Hardware Synthesis and Implementation
5. Running Model on Hardware

The details of each of these is described in the following units.

### 4.1 Model Training

#### Tools & Software used:

Python, PyTorch, Brevitas, Numpy

#### 4.1.1 BREVITAS - By Xilinx

Brevitas is an open source PyTorch library by Xilinx for quantization-aware training. It is a general purpose library which can be used to target bit-constrained datapaths for all sorts of hardware devices. We decided to use this library for providing us parameters because we can be sure of the bitwidths of the activations. Otherwise, starting from normally trained PyTorch models, some extra time and effort is required to constrain activations using post-training quantization.

Model Training was done via Quantized-Aware Training by using a framework called Brevitas in int8 data type.

#### 4.1.2 Batch Norm layer

---

In hardware, Batch norm can be easily catered for in the conv layers due its linearity. If the biases of conv layers in enabled, the parameters of batch norm layer can be easily absorbed into the weights and biases of conv layers.

### 4.1.3 Networks Tested

#### 1. Lenet-MNIST

This network was trained to achieve 98% accuracy on MNIST. The main goal of the training was not to get a higher accuracy but to increase the FPS while getting the same result as the GPU/CPU models. The network structure is as follows:

Layer	Stride	Neurons
3x3 Conv (same) + ReLU	1	
2x2 Maxpool	2	
3x3 Conv (same) + ReLU	1	
2x2 Maxpool	2	
Fully Connected layer		500
Fully Connected layer		10

Table 3 - Lenet-MNIST Structure

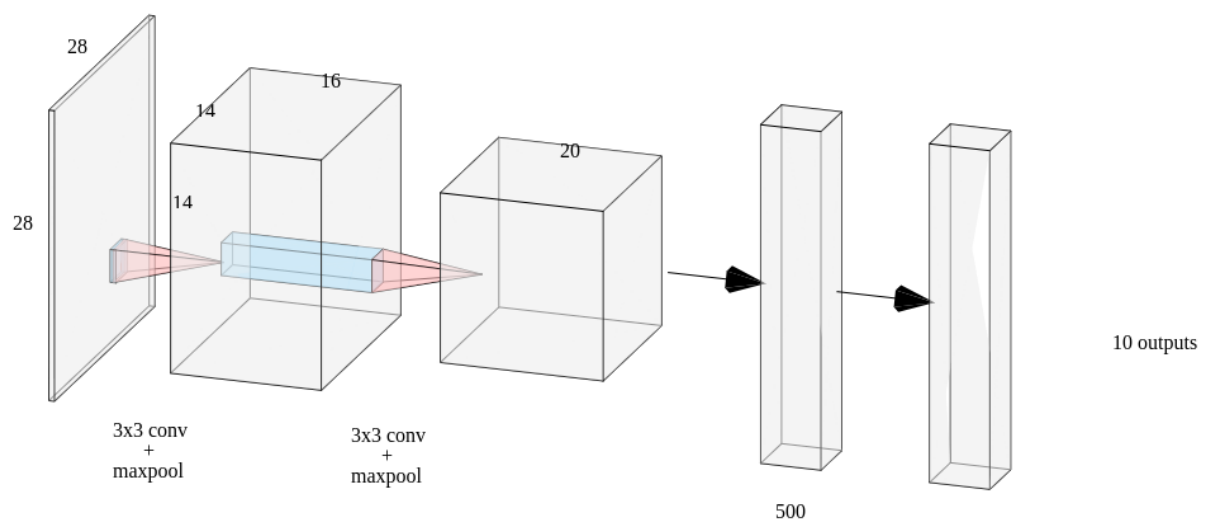


Figure 26 - Lenet-MNIST diagram

## 2. CNV - CIFAR10

This network is trained on CIFAR10 dataset with 84.12% accuracy. After quantization to int8 parameters the accuracy of 84.12% is retained. The structure is as follows:

Layer	Stride	Neurons
3x3 Conv (same) + ReLU	1	
2x2 Maxpool	2	
3x3 Conv (same) + ReLU	1	
2x2 Maxpool	2	
3x3 Conv (same) + ReLU	1	
2x2 Maxpool	2	
Fully Connected layer		512
Fully Connected layer		512
Fully Connected layer		10

Table 4 - CIFAR10-CNV Structure

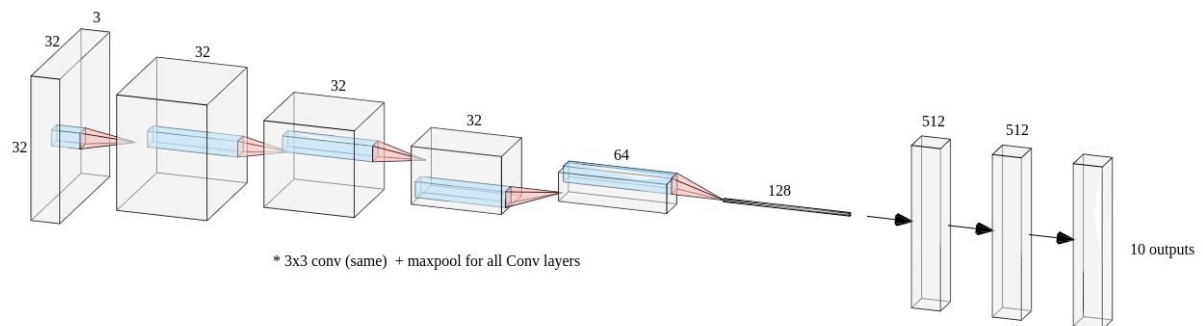


Figure 27 - CIFAR10-CNV diagram

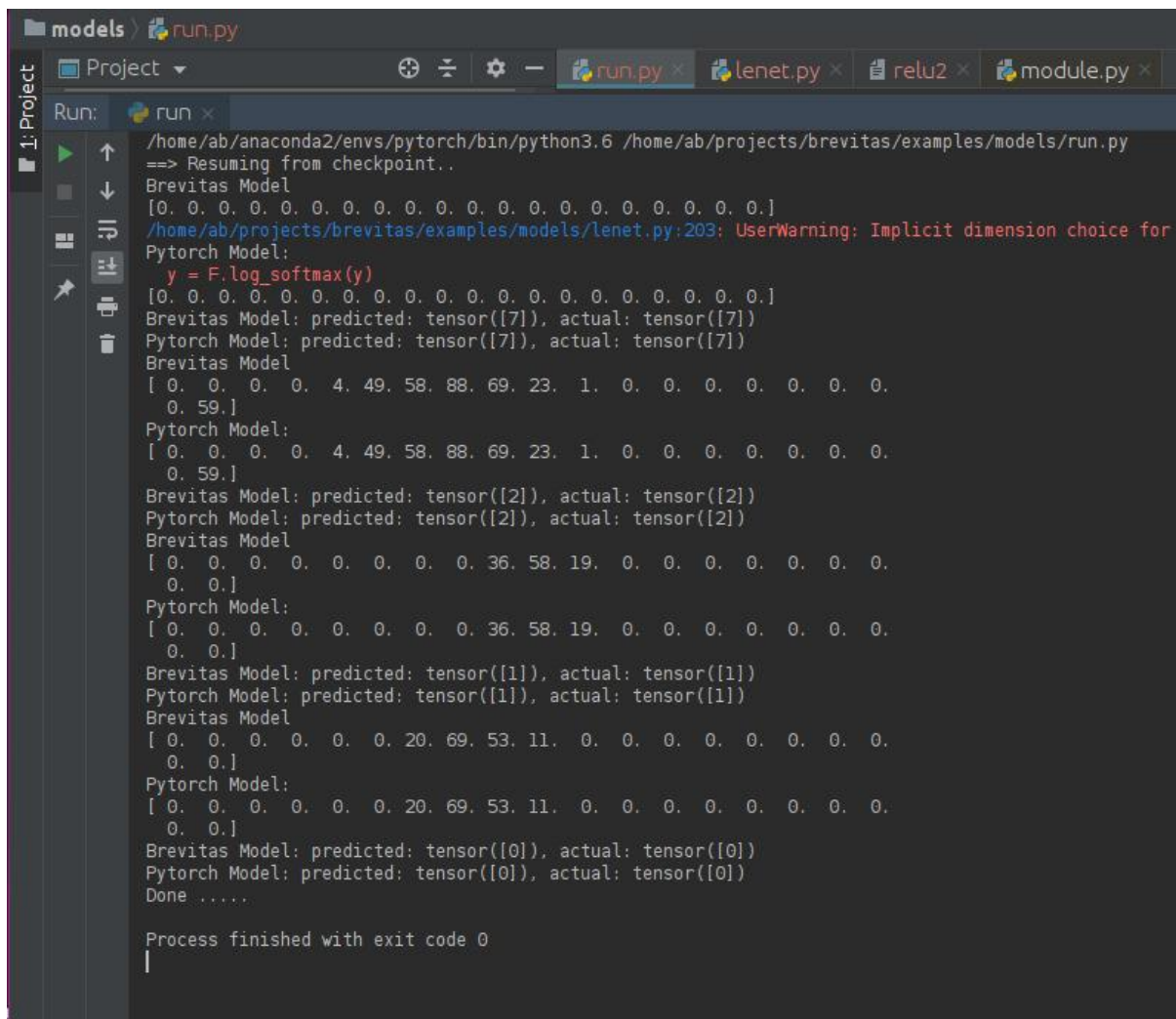
## 4.2 Software Simulation

### Tools & Software used:

Python, PyTorch and Numpy

The Brevitas library works in fixed point domain. Our goal was to understand its inner workings so that we can replicate these in Vivado-HLS. Also, the algorithm used for convolution is the standard one while we were implementing hardware design for Winograd convolution. This led to some challenges which are described in detail in “Challenges” section. For this purpose, we made PyTorch models identical to the Brevitas models. We loaded the PyTorch model with integer parameters and ran the model with appropriate shifts before running each layer. In this way, we knew exactly the operations needed in our hardware design. Now, we were ready to move to hardware implementation.

We got the same predictions using both models. A screenshot is attached which compares the final outputs obtained using both models for a number of random inputs.



```

models > run.py
Project
Run: run x
/home/ab/anaconda2/envs/pytorch/bin/python3.6 /home/ab/projects/brevitas/examples/models/run.py
==> Resuming from checkpoint..
Brevitas Model
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
/home/ab/projects/brevitas/examples/models/lenet.py:203: UserWarning: Implicit dimension choice for
Pytorch Model:
y = F.log_softmax(y)
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
Brevitas Model: predicted: tensor([7]), actual: tensor([7])
Pytorch Model: predicted: tensor([7]), actual: tensor([7])
Brevitas Model
[ 0.  0.  0.  0.  4. 49. 58. 88. 69. 23.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
Pytorch Model:
[ 0.  0.  0.  0.  4. 49. 58. 88. 69. 23.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
Brevitas Model: predicted: tensor([2]), actual: tensor([2])
Pytorch Model: predicted: tensor([2]), actual: tensor([2])
Brevitas Model
[ 0.  0.  0.  0.  0.  0.  0.  0. 36. 58. 19.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
Pytorch Model:
[ 0.  0.  0.  0.  0.  0.  0.  0. 36. 58. 19.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
Brevitas Model: predicted: tensor([1]), actual: tensor([1])
Pytorch Model: predicted: tensor([1]), actual: tensor([1])
Brevitas Model
[ 0.  0.  0.  0.  0.  0.  0.  0. 20. 69. 53. 11.  0.  0.  0.  0.  0.  0.  0.  0.]
Pytorch Model:
[ 0.  0.  0.  0.  0.  0.  0.  0. 20. 69. 53. 11.  0.  0.  0.  0.  0.  0.  0.  0.]
Brevitas Model: predicted: tensor([0]), actual: tensor([0])
Pytorch Model: predicted: tensor([0]), actual: tensor([0])
Done .....

Process finished with exit code 0

```

Figure 28 - Software Simulation

---

### 4.2.1 Weight Packing

The packing of weight is done in Python using a script. It needs to passed the structure of the entwork along with required SIMD and PE for each conv layer. This script produces weights packed in such an order that ensures high spacial locality.

## 4.3 Hardware Design

### Tools & Software used:

C++ & Vivado-HLS

### 4.3.1 Stage 1: C++ design with floating point operations

In the beginning, our goal was to produce a modularized working design in C++ for all operations required. All parameters and variables were in floating points. This is the first step when making an HLS design from scratch. After removing all the bugs, we were ready to synthesize the design for hardware. Although the operations were floating point, we still synthesized the design. The resources were too much, as expected.

### 4.3.2 Stage 2: Non-dataflow hardware design with fixed point operations

After successfully implementing the model with floating point parameters, we reduced bit widths until we started getting errors. Our next goal was to identify the sources of these errors and correct them. This step took us a lot of time. We had to move back and forth between C++ and PyTorch implementations to check output after each operation. We identified some problems and constraints as a result of using Winograd convolution which are described in detail in “Challenges” chapter.

After eradicating all errors and getting exactly the same result as expected, we synthesized the design to check its viability in hardware. We used Pynq-Z1 board as our platform. We Lenet model with two conv layers and two fully connected layers for our initial experiments because of small size and faster debugging. The details of the model are provided in the “Networks Tested” section. The results of the synthesis are as follows:

---

---

	<b>BRAM</b>	<b>DSP</b>	<b>FF</b>	<b>LUT</b>
<b>Used</b>	12	1820	49125	111154
<b>Available</b>	140	220	106400	53200
<b>Utilization</b>	8.5%	827%	46.1%	209%

Table 5 - Unoptimized Utilization

**Calculated FPS:** ~400 images/sec

At this moment, all the computation in convolution layers was not parameterized and not completely unrolled.

### 4.3.3 Final Stage: Dataflow design with integer operations

This design has been highly influenced by [10] and [11]. We reused a number of their building blocks and modified many.

#### 4.3.3.1 Custom DMA

We made a light-weight custom DMA which transfers the required number of blocks from DDR to on-chip memory. We reused the design of the DMA available in Xilinx's BNN repository with slight modification.

#### 4.3.3.2 Instruction Level parallelism

To benefit most from the dataflow design, each loop within the design needed to be pipelined with an initiation interval(II) of one [12, p. 902]. This was a difficult step for many blocks. The design underwent a number of iterations to achieve the desired initiation interval.

---



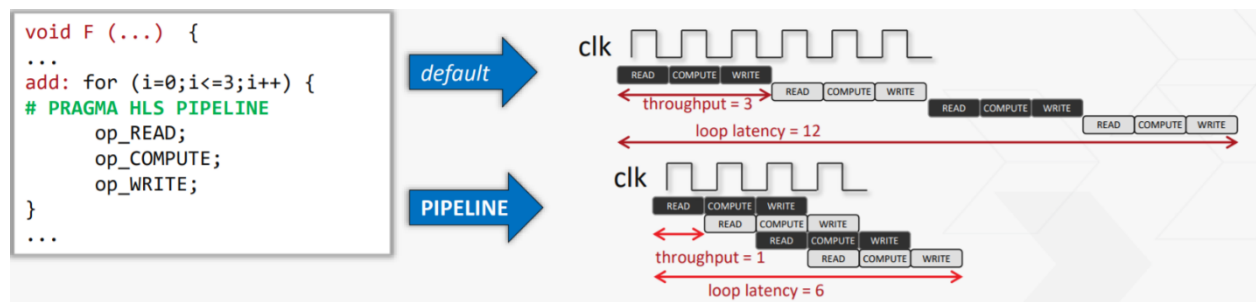


Figure 29 - Instruction Level Parallelism

### 4.3.3.3 HLS Streaming interface

In a neural network, a huge portion of memory is consumed to store the output result of a layer, called activation map. Without a streaming design, these activation maps need to be stored on-chip before being utilized by the next layer. The computation of the next layer does not start until all the computation of the previous layer had completed. This adds to the resource requirement and latency of the design.

To overcome this hurdle, we used a dataflow design. This design allows task-level parallelism between different functions [13]. For this purpose, the blocks need to be modified to operate on just a patch of input instead of the whole feature map.

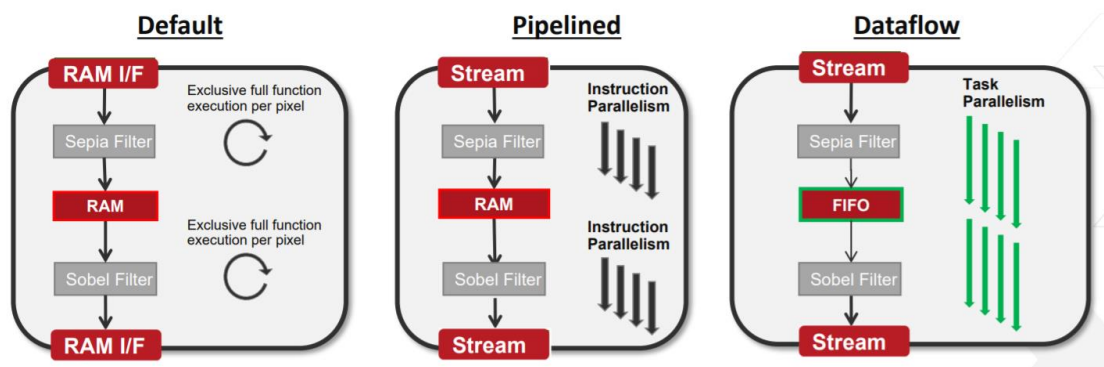


Figure 30 - Different Types of Hardware Designs

	Default	Pipelined	Dataflow
BRAM	2792	2790	24
FF	891	1136	883
LUT	2315	2114	1606
Interval (II)	128,744,588	4,150,224	2,076,613

Figure 31 - Specifications of Different Types of Hardware Designs

#### 4.3.3.4 Parameterizable Computation Engine

In the stage 2 of our design process, we fully unrolled the computation engine of both layers leading to huge DSP slice consumption and making the design impossible to be implemented on Pynq-Z1.

The next stage was to make the computation engine of each layer parameterizable to use as many resources as available on the board. We used two parameters to determine the degree of unrolling.

- i) Number of outputs to compute in parallel (PE)
- ii) Number of inputs to operate on in parallel (SIMD)

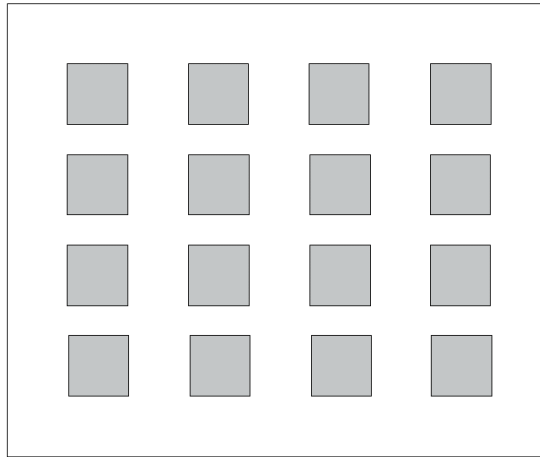


Figure 32 - Basic Compute Engine

Because of the nature of the Winograd convolution, we need to deal with a 4x4 tile instead of individual elements. So 4x4 PE engines are the minimum to be used for any layer (i.e. 16 DSP blocks are minimum to implement a layer). To operate on 2 inputs in parallel, we can increase SIMD to 2. This would require 32 DSP blocks.

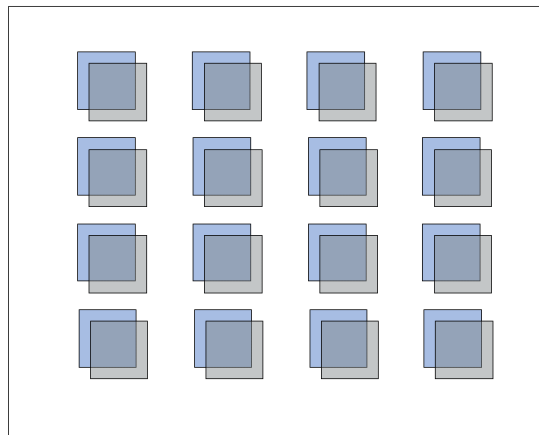


Figure 33 - Compute Engine with 2 SIMD lanes

To further extend this and produce output for 4 filters simultaneously, we can set  $PE=4$  and  $SIMD=2$ . This would require a total of 128 DSP blocks ( $16 \times 2 \times 4$ ).

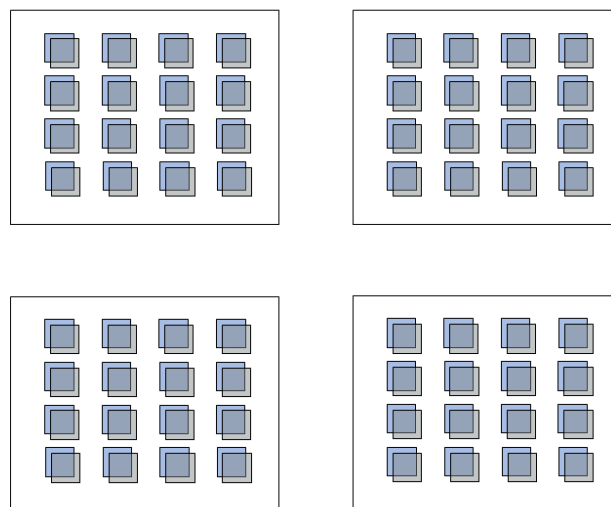


Figure 34 - Compute Engine with 2 SIMD lanes and computes 4 outputs in parallel

This would allow the layer to produce 4 output maps in parallel by sharing the inputs.

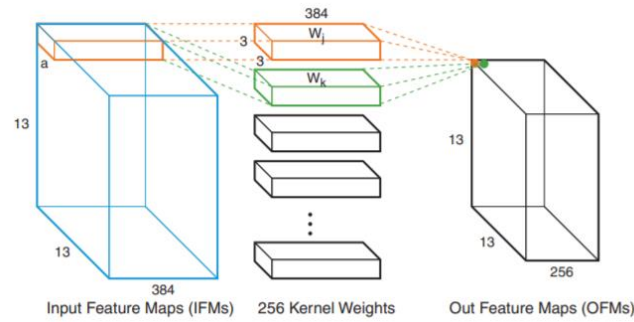


Figure 35 - Input Sharing [8]

These DSP blocks are just an estimation. DSP blocks of Xilinx 7000 series devices are capable of performing two int8 MACC operations simultaneously on one DSP block by taking advantage of its 18x27 bit multiplier. The result of DSP is looped back for accumulation making it efficient.

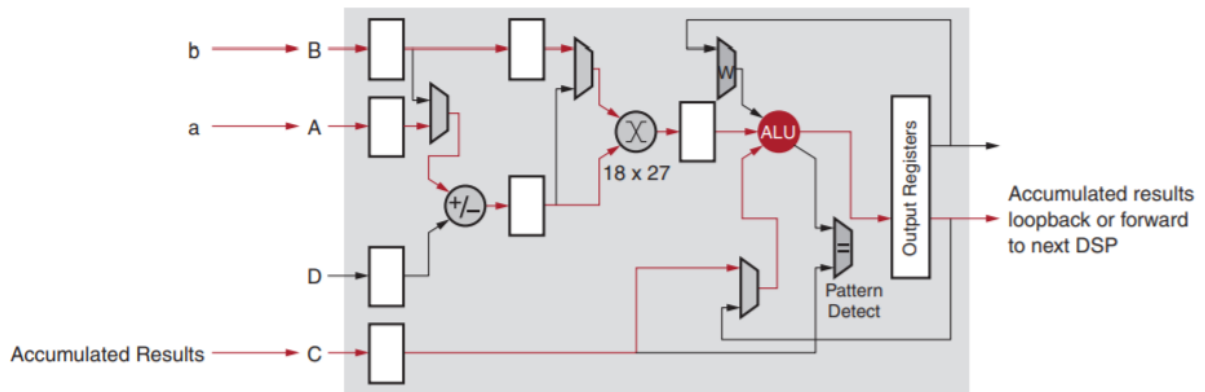


Figure 36 - DSP Slice with MACC mode [8]

This allows the Xilinx devices to achieve a 1.75 times performance improvement over other vendors.

#### 4.3.4 Vivado Cosimulation

Cosimulation in Vivado is a unique feature of HLS which is one of the reasons of its wide popularity. It allows the user to simulate the generated IP hardware without the need to

generate a separate testbench. It translates the C++ testbench into Verilog or VHDL and compares the outputs of the C++ function and the generated IP. If the output matches, the designed IP is functionally correct and we can move ahead.

The cosimulation also gives the clock cycles for the IP to finish operation. This is the exact number of clock cycles the end product will take. So, we can find the latency and throughput of the IP without having to implement the whole design.

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	84	85	4764	85	85	90

Figure 37 - RTL Co-simulation output



Figure 38 - Co-simulation waveforms

So the IP for our neural network is generated now with appropriate interfaces to communicate with other systems.

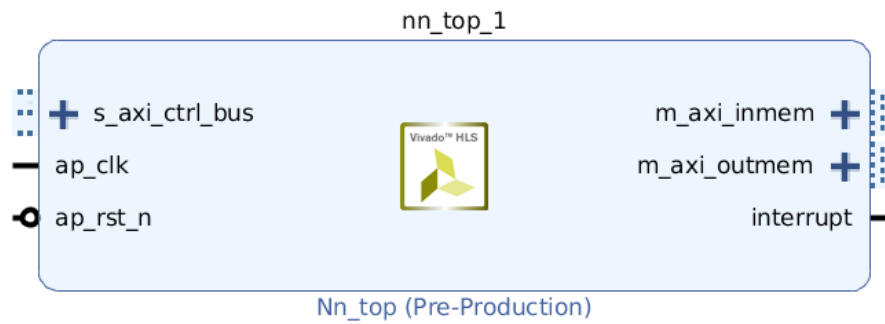


Figure 39 - Deep Neural Network IP

## 4.4 Hardware Synthesis and Implementation

### Tools & Software used:

Vivado Design Suite, Xilinx SDK & Pynq Linux

The work is not completed with Vivado-HLS but we have just generated the IP block design. The resource and timing estimates generated by Vivado-HLS are often more than actually required. To integrate the IP within the system and connect with PS, we use Vivado Design Suite's graphic user interface. We place the generated IP and Zynq PS and connect them appropriately through AXI interconnects. The final block diagram of the system is as follows:

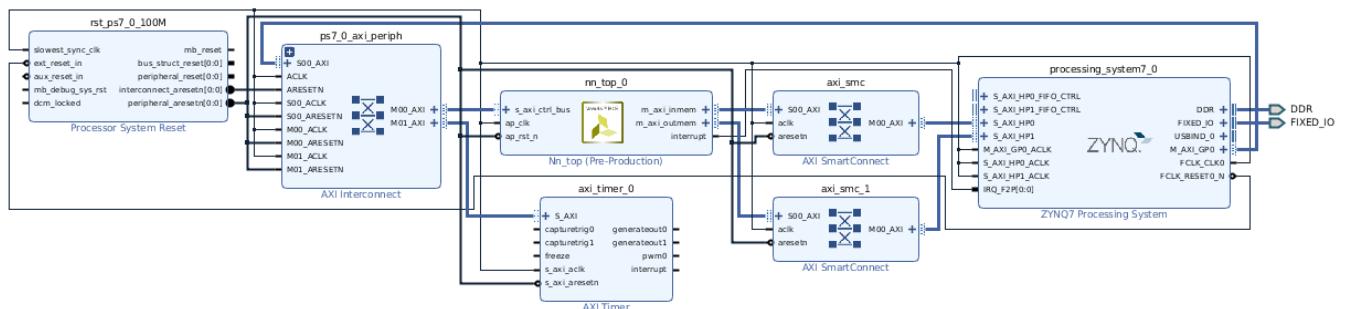


Figure 40 - System Block Design

The design then goes through the process of synthesis, place and route and bitstream generation. The actual timing and resource utilization is known at the end of the implementation step.

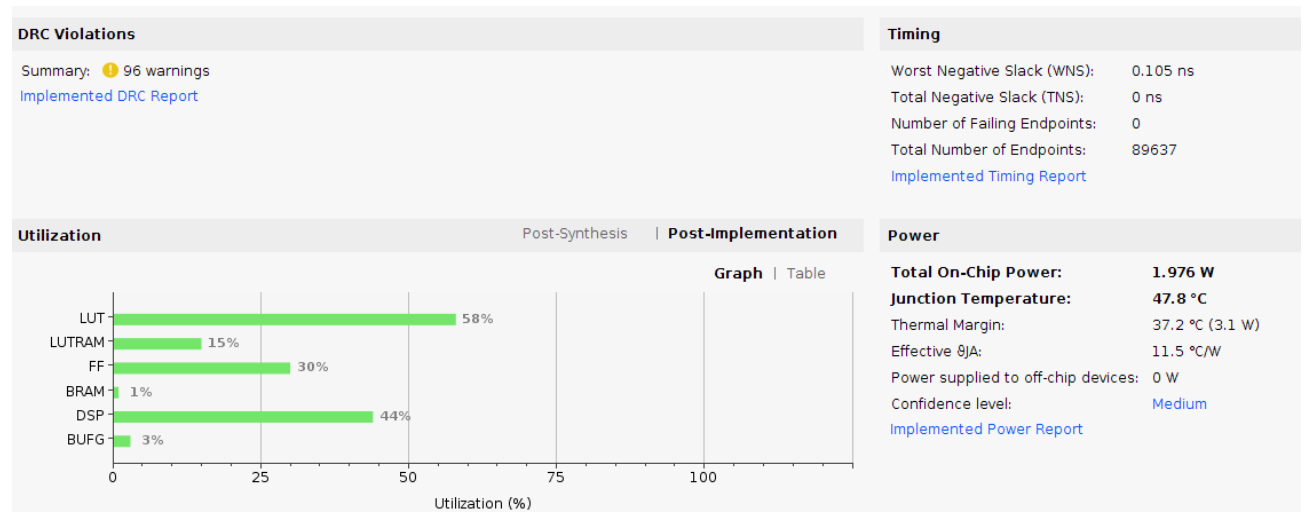


Figure 41 - Implementation Summary

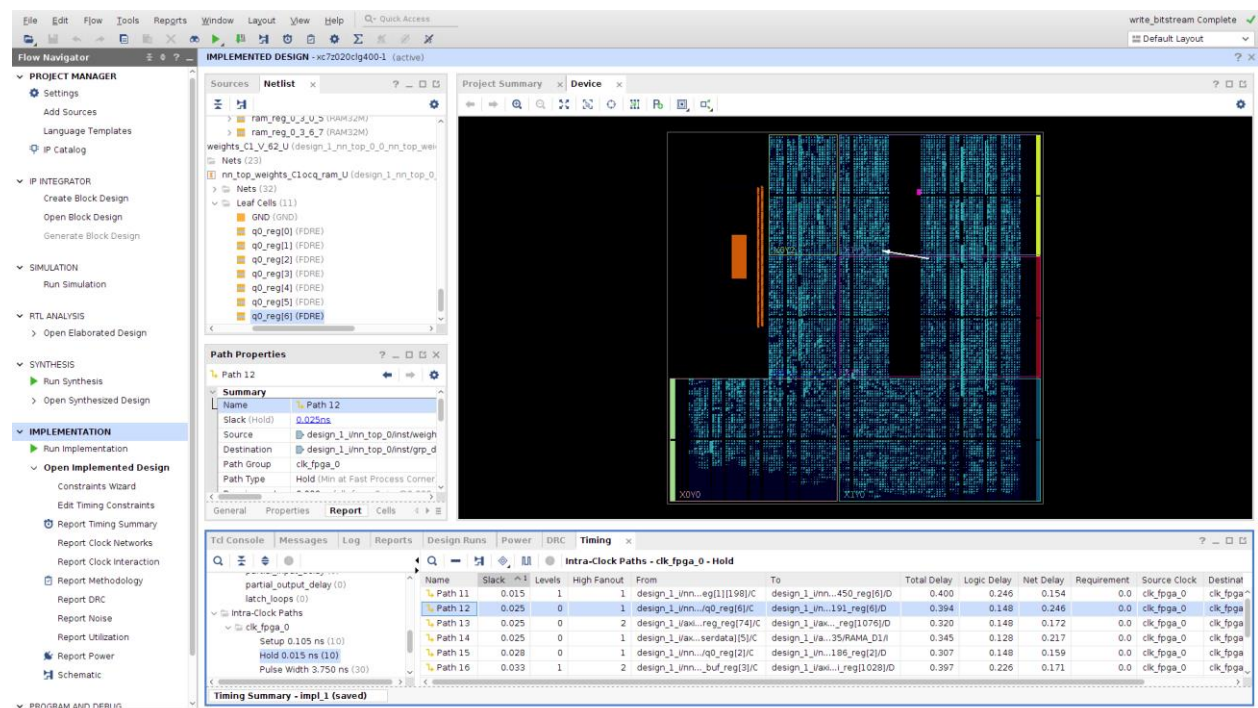


Figure 42 - Implementation Result

If the design fails to meet the timing, we can view the failing paths in the implemented design. Then we need to go back to HLS and make appropriate changes. Constraints can also be introduced during Vivado synthesis step but this requires some knowledge and experience. It is recommended not to include unnecessary constraints. This can lead to your design not working on hardware, as this happened to us and we lost about two weeks trying to debug the problem.

## **4.5 Running Models on Hardware**

### **Tools & Software used:**

Xilinx SDK, Pynq linux, Pynq-Z1 Board

There were two ways the implemented design could be tested on actual board:

1. Xilinx SDK
2. Linux

### **4.5.1 Xilinx SDK**

Vivado has this neat feature of producing drivers for the IPs in the block design. We can use these drivers and make a bare metal application. We would just have to use prebuilt functions and addresses already generated by Vivado Design Suite. This is a quick way to test if our hardware design works on the board. We can view the serial output using console in the Xilinx SDK environment.

We setup interrupts and tested the outputs of the hardware against the software outputs. There needs to be a 100% match because the RTL simulation was also producing 100% correct result.

---



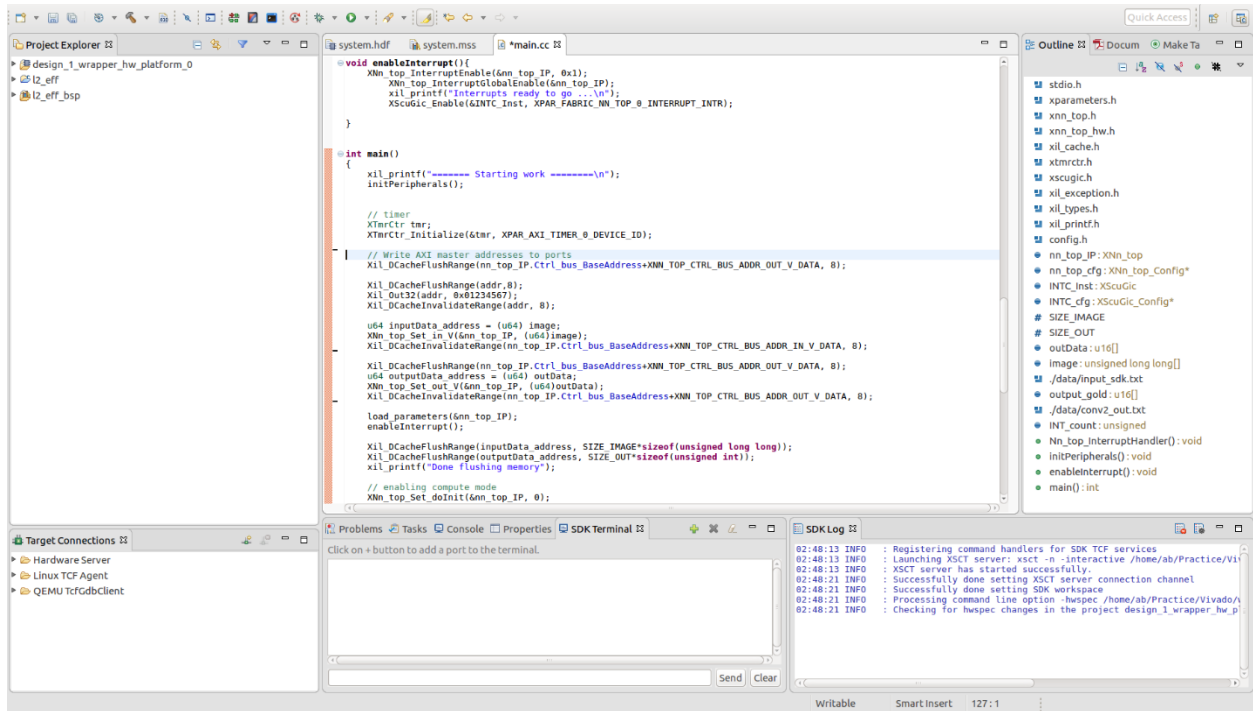


Figure 43 - Xilinx SDK Result

## 4.5.2 PYNQ Linux

We chose to use PYNQ Linux [14] to test out hardware design as well to provide a user friendly interface. We can use jupyter notebooks and Python to program FPGA and control the hardware IP. But for PYNQ Linux, we need to write our own drivers. We wrote the drivers for the Lenet-MNIST network and ran it on FPGA using PYNQ linux.

## 2. Hardware Inference

First of all a classifier needs to be instantiated. Using the LfcClassifier will allow to classify MNIST formatted images utilizing LFC network. There are two different runtimes available: hardware accelerated and pure software environment.

Once a classifier is instantiated the inference on MNIST images can be started using `classify_mnist` or `classify_mnists` methods - for both single and multiple images.

### Case 1:

#### *W1A1 - 1 bit weights and 1 bit activation*

```
In [3]: lfcW1A1_classifier = bnn.LfcClassifier(bnn.NETWORK_LFCW1A1, "mnist", bnn.RUNTIME_HW)
        lfcW1A1_classifier.classes

Out[3]: ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']

In [4]: result_W1A1 = lfcW1A1_classifier.classify_mnists("/home/xilinx/jupyter_notebooks/bnn/t10k-images-idx3-ubyte")

Inference took 28040.00 microseconds, 2.80 usec per image
Classification rate: 356633.39 images per second
```

Figure 44 - PYNQ Notebook Sample

We found the process of running of model on PYNQ linux too time consuming for quick development, so we did not use it for any other model.

# Chapter 5 Challenges

## 5.1 Training

Training of quantized neural networks is more time consuming than standard training. Doing this manually takes some experience and knowledge. In this regard, we started with using custom scripts for training.

Later, when we found out about Brevitas, we switched to it as it is supported by Xilinx. This made the process of training the quantized neural networks much easier.

For post-training quantization, we wrote our own scripts to constrict the weights and activations to 8 bits. The accuracy drop is minimal as explained in the “Post Training Quantization” section of “Literature Review” Chapter.

## 5.2 Challenges with Winograd

### 5.2.1 Precision

If the weights are quantized to 8-bit before Winograd transformation, the required bit width increases.

The worst case ranges and bit-widths of  $G^*g^*G^T$  are as follows respectively.

$$\begin{bmatrix} 1020 & 1530 & 1530 & 1020 \\ 1530 & 2295 & 2295 & 1530 \\ 1530 & 2295 & 2295 & 1530 \\ 1020 & 1530 & 1530 & 1020 \end{bmatrix}, \begin{bmatrix} 11 & 12 & 12 & 11 \\ 12 & 13 & 13 & 12 \\ 12 & 13 & 13 & 12 \\ 11 & 12 & 12 & 11 \end{bmatrix}.$$

Keeping in view this limitation, we have increased our margin of error when comparing the int8 outputs of standard convolution and int8 implementation of Winograd convolution.

---

Libraries like Brevitas optimize int8 performance for the standard convolution. If the aforementioned bit widths are not allowed, this will result in increase in error. Efforts are underway to optimize int8 performance in the winograd domain such as winograd-aware quantized training [15].

### 5.2.2 Memory

$G^*g^*G^T$  transforms the filter  $g$  to the Winograd domain, matching the dimensions of the input tile. This results in an increase of run-time memory associated with the weights. As memory is an important resource in FPGA, this overuse needs to be kept in mind.

## 5.3 Over-utilization of Resources

In the initial designs when the compute engines were non-parameterizable, we had to experience over-utilization of resources. Inspired by the FINN design [11], we also made our compute engine parameterizable using PE and SIMD parameters.

More resources do not mean more speed. The pipeline needs to be balanced which is achieved by setting the right combination of PE and SIMD.

## 5.4 Control Sets

We struggled with managing number of control sets. We designed a number of designs with less resources than required to be placed on the board. But, the number of control signals were more than unique control sets available on PYNQ-Z1. As a result, we were not able to run those design on hardware.

This problem remains to be solved. One drawback of working with HLS is that we do not have control over the control signals. This problem can be solved in Vivado Design Suite using constraints which required SoC design experience.

---

## **5.5 Hardware Debugging**

There were some problems when running the design on hardware initially. We needed to debug the hardware design which is quite challenging.

For the purpose of debugging of actual hardware, we used the Xilinx's Integrated Logic Analyzer (ILA). This can be integrated into the block design or added to the design in the form of a constraint file. By using a constraint file, we get a finer control on which signals to monitor on hardware to debug the design.

The problem was traced to incorrect constraints.

---

## Chapter 6 Results and Discussion

### 6.1 Lenet – MNIST

With the same PE/SIMD configuration, following are different designs which can be obtained.

- Resource are percentage of PYNQ-Z1 board resources.
- Power is noted from the Vivado Implementation summary report.
- FPS are of obtained from actually running the design on hardware.

**Note:** FPS are less than desired because the design is pipelined but we only ran 1 image through it on hardware. When running multiple images, the FPS will almost be double

	LUT	LUTRAM	FF	BRAM	DSP	Power (W)	FPS
<b>Design 1</b>	75%	15.84%	33.55%	1.07%	46.36%	1.92	17.1k
<b>Design 2</b>	76.22%	15.84%	37.75%	21.43%	46.36%	1.97	22.2k
<b>Design 3</b>	58.21%	14.52%	29.55%	1.07%	43.64%		20.15k
<b>Design 4</b> (Xilinx reuse)	52.53%	14.99%	24.26%	3.57%	43.64%	1.87	7.2k

Table 6 - Lenet-MNIST Hardware Design Comparison

One of the main reasons of rapid adoption of HLS is due to its facility of exploring the design space quickly. We were also able to find different ways in we can trade-off speed for lesser resource utilization.

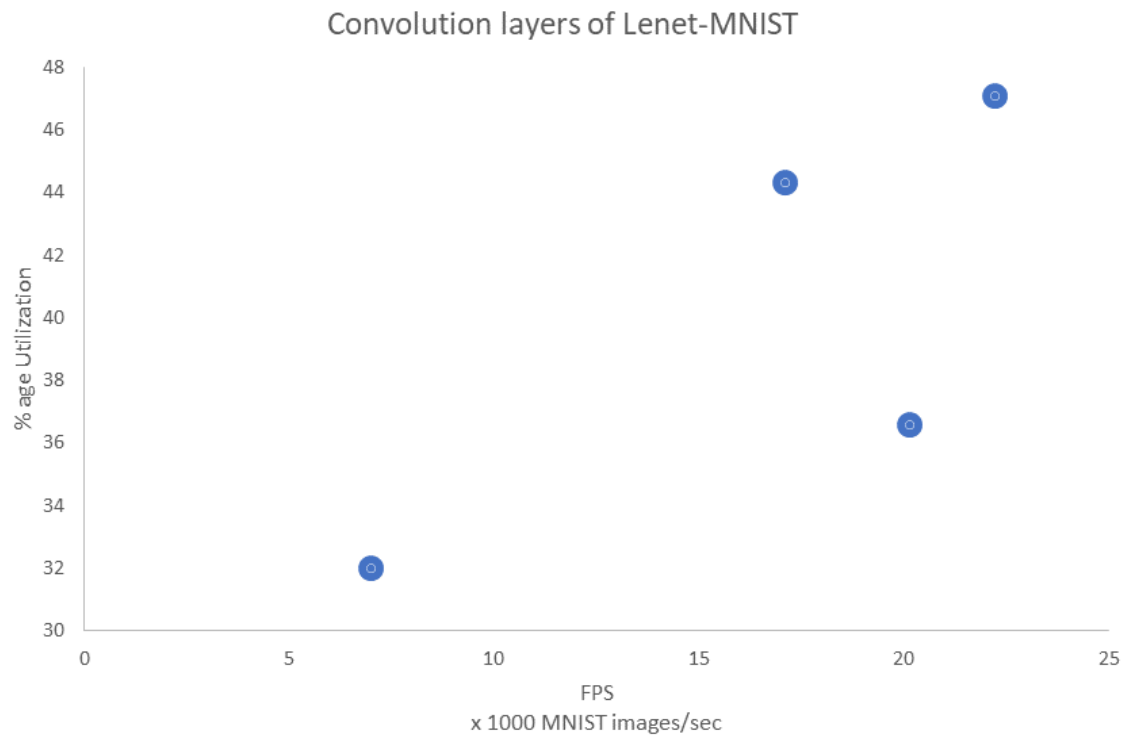


Figure 45 - Lenet-MNIST Design Space Exploration

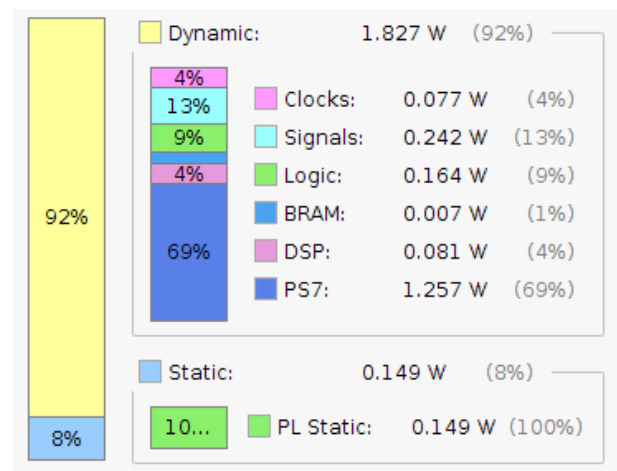


Figure 46 - Summary of Power Utilization of SoC

It can be observed that most of the power is being consumed by the PS while only 8.155% of the power is being consumed by the PL part.

Device	Frequency	FPS
Intel i7-7700HQ CPU	1093 MHz	6.6K
Tesla K80 (150 W)	562 MHz	3.5M
PYNQ-Z1	100 MHz	22.2K

Table 7 - Lenet-MNIST on different Platforms

It can be observed that the FPGA is much faster than CPU but still slower than GPU. We could only map the following configuration onto the FPGA. The resource constraints of PYNQ-Z1 forced us to not utilize the full ability of our design.

## 6.2 Cifar10- CNV

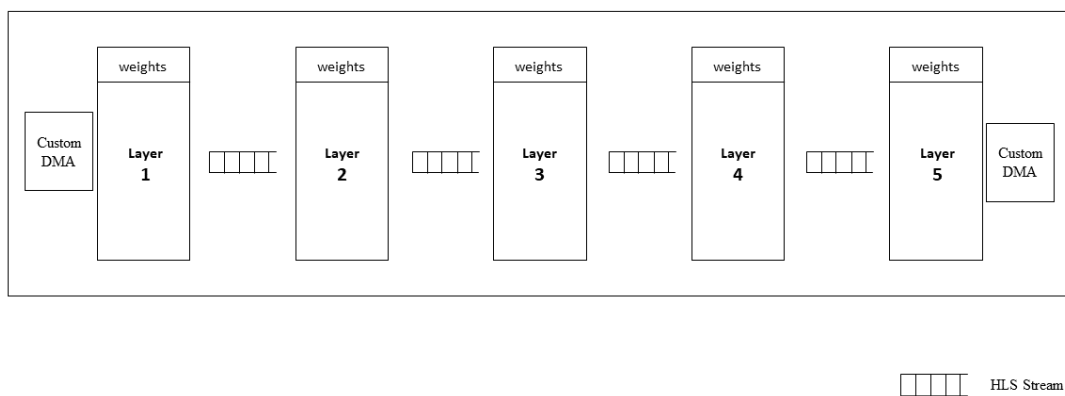


Figure 47 - CIFAR10-CNV Hardware Design

The results of the CNV model that we were able to fit on PYNQ-Z1 are as follows. All PEs and SIMDs are set to 1 i.e. minimum

	LUT	LUTRAM	FF	BRAM	DSP	Power (W)	FPS
<b>Design 4</b> (Xilinx reuse)	53.71%	5.03%	58.98%	85%	42.73%	1.94	573

Table 8 - CIFAR10-CNV Hardware Design Utilization



Comparison of FPGA design with CPU and GPU is as follows

Device	Frequency	FPS
Intel i7-7700HQ CPU	1093 MHz	6K
Tesla P100-16GB (250W)	1190 MHz	78.65K
PYNQ-Z1	100 MHz	573

Table 9 - CIFAR10-CNV on Different Platforms

It can be observed that FPGA speed is not comparable to others. It is due to insufficient resource. We were able to synthesize a design with 3K FPS on FPGA, but it had too many control sets to be placed onto the PYNQ-Z1. Due to the Covid-19 pandemic, this was the only hardware available to us.

### 6.3 CHaiDNN

ChaiDNN is originally supported for MPSoC. We ported it to SoC and ran some example networks using this flexible generic architecture. Following results were obtained on ZC706.

Network	Dataset	Accuracy (Top 1%)	FPS (on ZC706)
Googlenet-8bit	Imagenet	67.09 %	8.973396

Table 10 - CHaiDNN Results on ZC706

---

## Chapter 7 Conclusion

We were successful in completing a Library with user-friendly interface with a different type of compute engine that is not open source yet in the world of HLS. Using this library, we were able to test a number of configurations in matters of minutes. This is in contrast to conventional RTL design where it takes weeks to months to test different designs.

We have achieved all of our objectives of designing a complete flow of deployment of neural networks from training to inference on FPGA. Due to the efficiency of FPGA for inference, these are the platforms of choice for edge devices.

Our design is not open source yet due to possibility of publication. When it is open sourced, It can be found at [16].

### 7.1 Future Work

This is a relatively new area with a lot of interests from big organizations like Nvidia, Mentor Graphics and Xilinx. There are a number of areas to explore including:

- Generic hardware accelerator
  - CHaiDNN for smaller boards
  - Recurrent Neural Networks and LSTMs
  - New version of FINN library
  - Using Chisel for hardware design
-

# Appendix

## i) Github Repositories

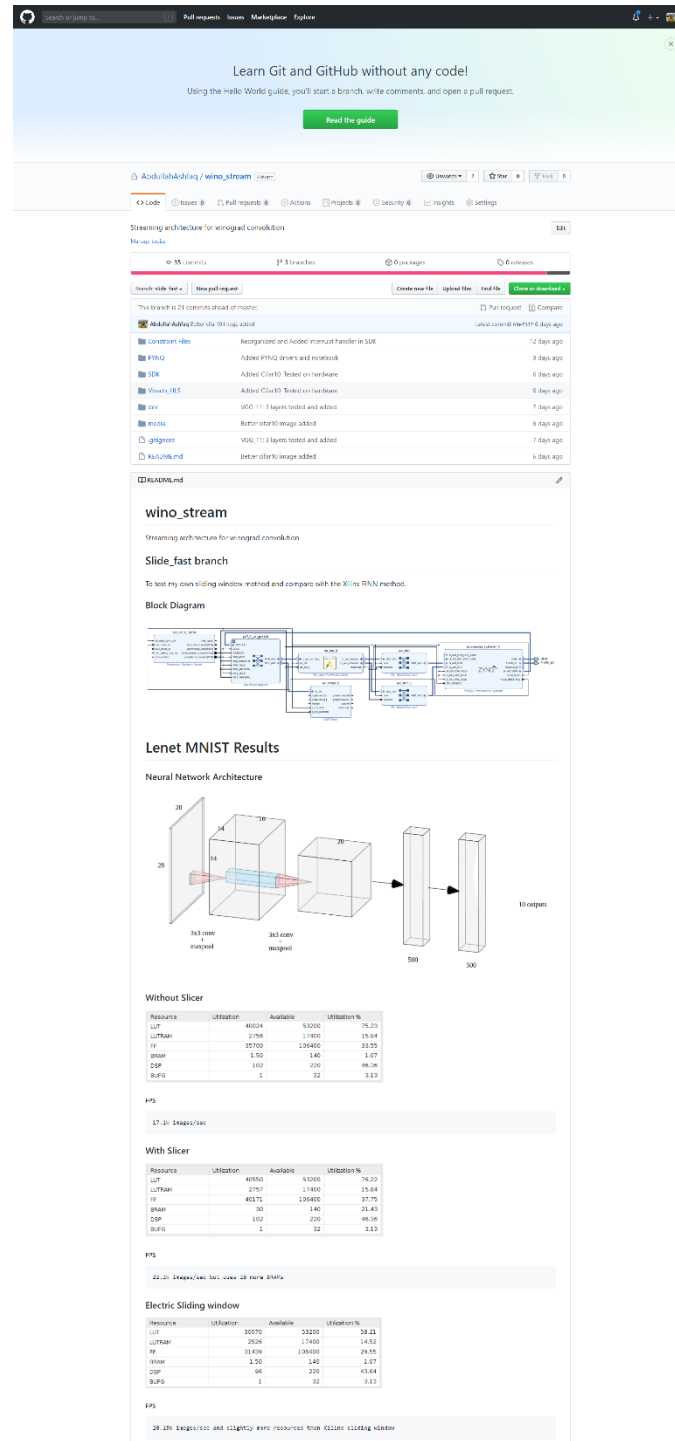


Figure 48 - Github Repository

## ii) Sample PYNQ driver

```

class nn_top_Driver:
    def __init__(self, bit_path=None):
        self.overlay = Overlay(bit_path)
        self.IP = self.overlay.nn_top_0
        self.reg = self.IP.register_map
        self.base_addr = self.IP.mmio.base_addr
        #Input and Output buffers
        self.input_buffer = None
        self.output_buffer = None

    def run(self):
        self.reg.CTRL.AP_START = 1
        while (not self.reg.CTRL.AP_DONE):
            pass

    def initialize(self, file, targetmem, CH, ROW, COL, isbias):
        if(isbias):
            array = np.loadtxt(file, dtype=ctypes.c_longlong, delimiter=',', usecols=[0])
        else:
            array = np.loadtxt(file, dtype=ctypes.c_uint32, delimiter=',',
                               usecols=[0],
                               converters={_:lambda s: int(s, 16) for _ in range(CH*ROW*COL)})

        array = array.reshape([CH, ROW, COL])
        print('Writing parameters...')
        for ch in range(CH):
            for row in range(ROW):
                for col in range(COL):
                    print('ch:{}, row:{}, col:{}'.format(str(ch), str(row), str(col)))
                    self.reg.doInit = 1
                    self.reg.targetmem = targetmem
                    self.reg.target_ch = ch
                    self.reg.target_row = row
                    self.reg.target_col = col
                    self.reg.numReps = 1

                    if(isbias):
                        self.reg.val_V_1 = int(array[ch,row,col]) & (0xffffffff)
                        self.reg.val_V_2 = int(array[ch,row,col]) >> 32 & (0xffffffff)
                    else:
                        self.reg.val_V_1 = (array[ch,row,col]) & (0xffffffff)
                        self.reg.val_V_2 = (array[ch,row,col]) >> 32 & (0xffffffff)

                    # garbage vals
                    self.reg.in_V_1 = 0
                    self.reg.in_V_2 = 0
                    self.reg.out_V_1 = 0
                    self.reg.out_V_2 = 0
                    # Run IP
                    self.run()
                    self.reg.doInit = 0

```

Figure 49 - PYNQ Driver

### iii) Sample HLS top function

```
Mem2Stream<...>(in, inStream_0, reps);
StreamingDataWidthConverter_Batch<...>(inStream_0, inStream_1, reps);

Streaming_pad<...>(inStream_1, c0_in_padded, reps);
convlayer<...>(c0_in_padded, c0_outstream, weights_C0, bias_C0, reps);

Streaming_pad<...>(c0_outstream, c1_in_padded, reps);
convlayer<...>(c1_in_padded, c1_outstream, weights_C1, bias_C1, reps);

Streaming_pad<...>(c1_outstream, c2_in_padded, reps);
convlayer<...>(c2_in_padded, c2_outstream, weights_C2, bias_C2, reps);

Streaming_pad<...>(c2_outstream, c3_in_padded, reps);
convlayer<...>(c3_in_padded, c3_outstream, weights_C3, bias_C3, reps);

Streaming_pad<...>(c3_outstream, c4_in_padded, reps);
convlayer<...>(c4_in_padded, c4_outstream, weights_C4, bias_C4, reps);

StreamingDataWidthConverter_Batch<...>(c4_outstream, outstream, reps);
Stream2Mem<...>(outstream, out, reps);
```

### iv) Sample SDK function

```
#include <stdio.h>

// driver files for IP
#include <xparameters.h>
#include "xnn_top.h"
#include "xnn_top_hw.h"
#include <xil_cache.h>
```

```
// driver for our axi timer
#include <xtmrctr.h>

// driver for Interrupt Controller
#include <xscugic.h>
#include <xil_exception.h>

// Extra utilities
#include <xil_types.h>
#include <xil_printf.h>

// Loading configurations
#include "config.h"

// Creating accelerator object
XNn_top nn_top_IP;
XNn_top_Config *nn_top_cfg;

// Input and Output Arrays
#define SIZE_IMAGE 98
#define SIZE_OUT 490

u16 outData[SIZE_OUT] = {0};
unsigned long long image[SIZE_IMAGE] = {
#include "./data/input_sdk.txt"
};

u16 output_gold[SIZE_OUT]={
#include "./data/conv2_out.txt"
};

void initPeripherals(){
    xil_printf("initializing Nn_top\n");
    int status = 0;
    nn_top_cfg = XNn_top_LookupConfig(XPAR_NN_TOP_0_DEVICE_ID);
    if(nn_top_cfg)
    {
        status = XNn_top_CfgInitialize(&nn_top_IP, nn_top_cfg);
        if(status != XST_SUCCESS)
        {
            xil_printf("Error initializing XNn_top IP\n");
        }
    }
}

int main()
{
    initPeripherals();

    // timer
    XTmrCtr tmr;
```

```
XTmrCtr_Initialize(&tmr, XPAR_AXI_TIMER_0_DEVICE_ID);

load_parameters(&nn_top_IP);
enableInterrupt();

Xil_DCacheFlushRange(inputData_address, SIZE_IMAGE*sizeof(unsigned long long));
Xil_DCacheFlushRange(outputData_address, SIZE_OUT*sizeof(unsigned int));

// enabling compute mode
XNn_top_Set_dolnit(&nn_top_IP, 0);
// no. of test examples
XNn_top_Set_numReps(&nn_top_IP, 1);

// passing address of input and output buffer
XNn_top_Set_in_V(&nn_top_IP, (u64)image);
XNn_top_Set_out_V(&nn_top_IP, (u64)outData);

// Start Timer
XTmrCtr_Reset(&tmr,0);
int tick1 = XTmrCtr_GetValue(&tmr,0), tick2;
XTmrCtr_Start(&tmr,0);

XNn_top_Start(&nn_top_IP);
while(!XNn_top_IsDone(&nn_top_IP));

// stop the timer
XTmrCtr_Stop(&tmr,0);
tick2 = XTmrCtr_GetValue(&tmr,0);

// Calculate Time
double time = (double)(tick2-tick1)/(double)XPAR_AXI_TIMER_0_CLOCK_FREQ_HZ;
printf("Time: %.9f ms\r\nFps: %.2f\r\n",time,1/time);

Xil_DCacheInvalidateRange(outputData_address, SIZE_OUT*sizeof(int16_t));

int diff=0;
for(int i = 0; i<SIZE_OUT; i++){
    xil_printf("Hardware[%d] = %d,\t gold[%d] = %d\n", i, outData[i], i, output_gold[i]);
    if(outData[i] != output_gold[i])
        diff++;
}

xil_printf("Total Differences: %d\n", diff);
return 0;
}
```

---

## Bibliography and Webography

- [1] “ZC706 Evaluation Board for the Zynq-7000 XC7Z045 SoC User Guide (UG954),” p. 103, 2019.
  - [2] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1,” *ArXiv160202830 Cs*, Mar. 2016, Accessed: Jun. 15, 2020. [Online]. Available: <http://arxiv.org/abs/1602.02830>.
  - [3] A. Lavin and S. Gray, “Fast Algorithms for Convolutional Neural Networks,” *ArXiv150909308 Cs*, Nov. 2015, Accessed: Jun. 15, 2020. [Online]. Available: <http://arxiv.org/abs/1509.09308>.
  - [4] L. Meng and J. Brothers, “Efficient Winograd Convolution via Integer Arithmetic,” *ArXiv190101965 Cs*, Jan. 2019, Accessed: Jun. 15, 2020. [Online]. Available: <http://arxiv.org/abs/1901.01965>.
  - [5] B. Barabasz, A. Anderson, K. M. Soodhalter, and D. Gregg, “Error Analysis and Improving the Accuracy of Winograd Convolution for Deep Neural Networks,” *ArXiv180310986 Cs Stat*, Mar. 2018, Accessed: Jun. 15, 2020. [Online]. Available: <http://arxiv.org/abs/1803.10986>.
  - [6] Y. Huang, J. Shen, Z. Wang, M. Wen, and C. Zhang, “A High-efficiency FPGA-based Accelerator for Convolutional Neural Networks using Winograd Algorithm,” *J. Phys. Conf. Ser.*, vol. 1026, p. 012019, May 2018, doi: 10.1088/1742-6596/1026/1/012019.
  - [7] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks,” p. 13.
  - [8] Y. Fu, E. Wu, A. Sirasao, S. Attia, K. Khan, and R. Wittig, “Deep Learning with INT8 Optimization on Xilinx Devices,” p. 11, 2017.
  - [9] R. Krishnamoorthi, “Quantizing deep convolutional networks for efficient inference: A whitepaper,” *ArXiv180608342 Cs Stat*, Jun. 2018, Accessed: Jun. 15, 2020. [Online]. Available: <http://arxiv.org/abs/1806.08342>.
-



- 
- [10] M. Blott, T. Preusser, N. Fraser, G. Gambardella, K. O'Brien, and Y. Umuroglu, "FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks," *ArXiv180904570 Cs*, Sep. 2018, Accessed: Jun. 15, 2020. [Online]. Available: <http://arxiv.org/abs/1809.04570>.
- [11] Y. Umuroglu *et al.*, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," *Proc. 2017 ACM SIGDA Int. Symp. Field-Programm. Gate Arrays - FPGA 17*, pp. 65–74, 2017, doi: 10.1145/3020078.3021744.
- [12] "Vivado Design Suite User Guide: High-Level Synthesis (UG902)," p. 573, 2018.
- [13] J. Myers, "HLS Tips and Tricks," p. 23, 2018.
- [14] *Xilinx/PYNQ*. Xilinx, 2020.
- [15] J. Fernandez-Marques, P. N. Whatmough, A. Mundy, and M. Mattina, "Searching for Winograd-aware Quantized Networks," *ArXiv200210711 Cs Stat*, Feb. 2020, Accessed: Jun. 13, 2020. [Online]. Available: <http://arxiv.org/abs/2002.10711>.
- [16] "AbdullahAshfaq/wino\_stream," *GitHub*.  
[https://github.com/AbdullahAshfaq/wino\\_stream](https://github.com/AbdullahAshfaq/wino_stream) (accessed Jun. 14, 2020).
-