Implementation of a Convolutional Neural Network in FPGA for Histopathologic Cancer Detection

Submitted in the fulfilment of requirement of

EE712 - Embedded Systems Design

By

Name : Pradumn Kumar	**Roll N.o** : 213070055
Name : Piyush Palav	**Roll N.o** : 213079017
Name : Shreyansh Neekhre	**Roll N.o** : 213079029

Under the Guidance of

Prof. Laxmeesha Somappa, Prof. Dinesh Sharma

E.E Department, IIT Bombay

Department of Electrical Engineering

Indian Institute of Technology Bombay

Contents

1. Introduction

Lymph nodes are small glands that filter the fluid in the lymphatic system and they are the first place a breast cancer is likely to spread. Histological assessment of lymph node metastases is part of determining the stage of breast cancer in TNM classification which is a globally recognized standard for classifying the extent of spread of cancer. The diagnostic procedure for pathologists is tedious and time-consuming as a large area of tissue has to be examined and small metastases can be easily missed which makes Machine Learning an ideal choice in terms of accuracy and ease of usability.

Convolutional Neural Networks (CNNs) are widely used in such Computer Vision problems over other types of neural networks, due to some of their following advantages:

1. Translation Invariance: CNNs are able to recognize objects in an image regardless of their position or orientation. This is because the convolution operation is translation invariant, which means that the same features are detected regardless of where they appear in the image.

2. Parameter Sharing: In CNNs, the same set of parameters (weights and biases) are used for each neuron in a layer, which greatly reduces the number of parameters that need to be learned. This not only makes the network easier to train but also helps to prevent overfitting and makes it easier to implement it in resource constrained systems.

Current hardware platforms for deep learning algorithms mainly consist of GPUs, ASICs, and FPGAs. One significant advantage of FPGAs over others is that they are highly programmable and reconfigurable, allowing for efficient customization of the hardware architecture for a given CNN model. This flexibility also enables FPGA-based systems to adapt to different input image sizes and data types, which is essential in medical image analysis, where different imaging modalities produce images of varying resolutions and formats. Additionally, FPGAs offer lower power consumption, shorter design times and have lower development costs which makes them an attractive choice for implementing CNNs in medical image analysis applications.

Nevertheless, implementing CNNs on FPGAs poses several challenges. One of the main challenges is the limited resources available on an FPGA. Additionally, the design of a CNN on an FPGA requires careful consideration of the available resources and the optimization of the design to make efficient use of those resources which can be a time-consuming and complex process.

Thus, our primary objective is to train our simple CNN Model on cloud, convert it into a hardware description language (HDL) using Vivado HLS and deploy it on FPGA for real-time inference on new cancer images.

2. Implementation

2.1 CNN Model Architecture

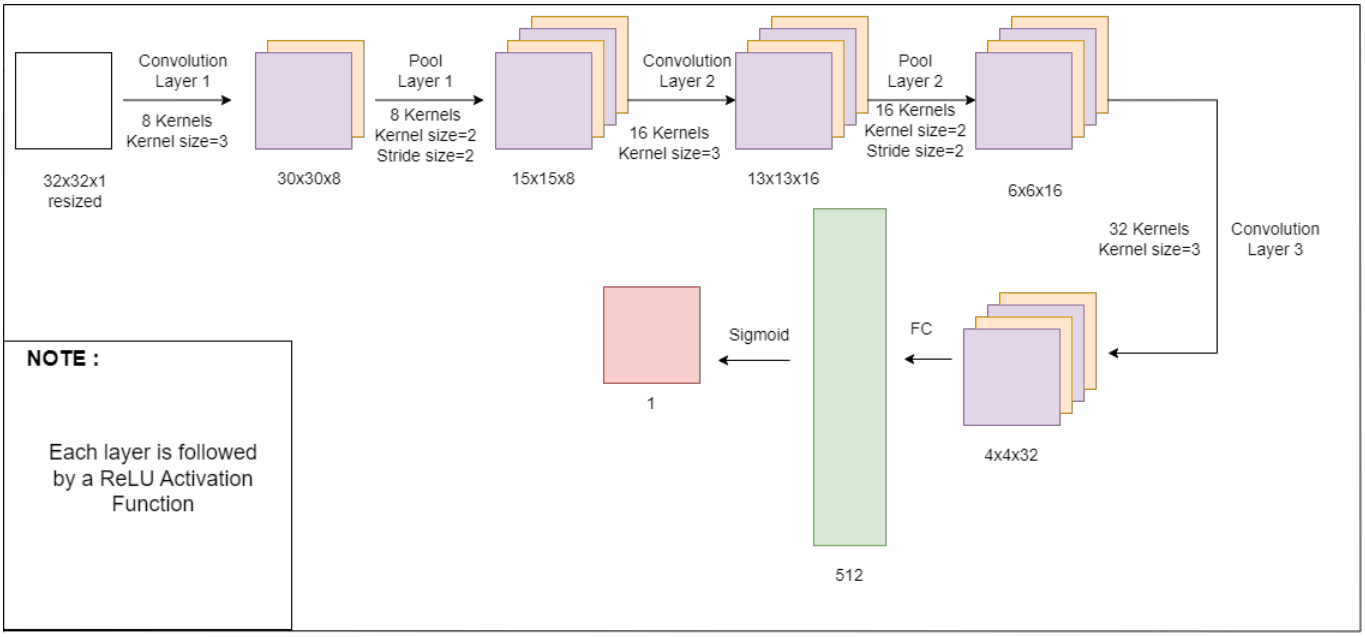
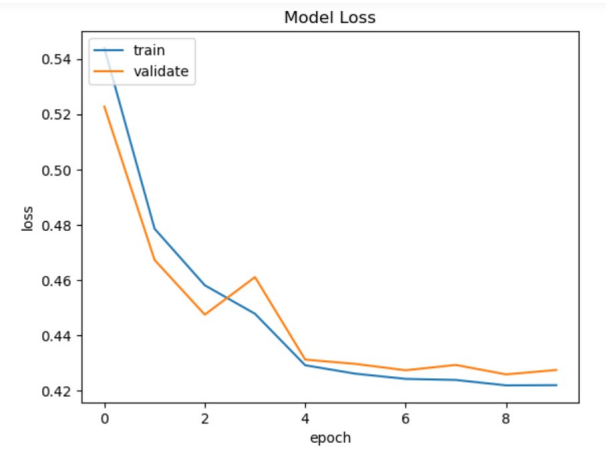


Figure 1: CNN Model Architecture

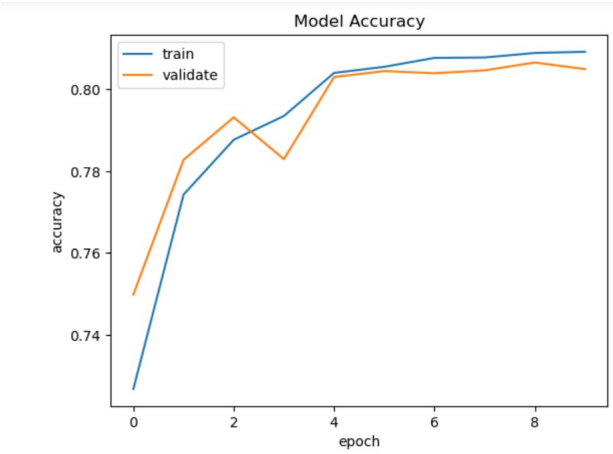
Original RGB image of 96x96 is converted to grayscale and rescaled to 32x32 to reduce feature map size of each layer which results in decreased training time as well as lower FPGA resource consumption. Max Pooling layer is added after each convolution layer to reduce the number of learnable parameters. Each convolution layer is followed by a ReLU activation layer and sigmoid is used as the activation function after the fully connected layer to predict whether the cancer is benign or malignant. Some of the other hyperparameters (except those that can be inferred from above architecture) chosen are as follows:

1. Train and Validation Mini Batch Size - 32
2. Number of epochs - 10 (validation accuracy saturates after 10)
3. Learning Rate - 0.001(initial) decreasing by a 0.1 factor on validation loss plateau
4. Learning rate optimization algorithm - Adam
5. Loss function - Binary Cross Entropy

At the end, a trade-off is made, to achieve 81% accuracy and AUC score of 0.89 while keeping total parameter count $\approx$ 6,400 to achieve low Pynq resource utilization.



Model Loss vs Epoch



Model Accuracy vs Epoch

2.2 Implementation in C++ using Vivado HLS

Inference step of CNN is written in C++ and synthesized in hardware description language (HDL) using Vivado HLS. Three main functions: conv(), pool() and fullyconnected() are used and hls::stream is used to pass data between functions. Since CNNs do not need the entire data from the previous layer to calculate output of the current layer, all layers can be executed in dataflow style by using #PRAGMA HLS DATAFLOW.

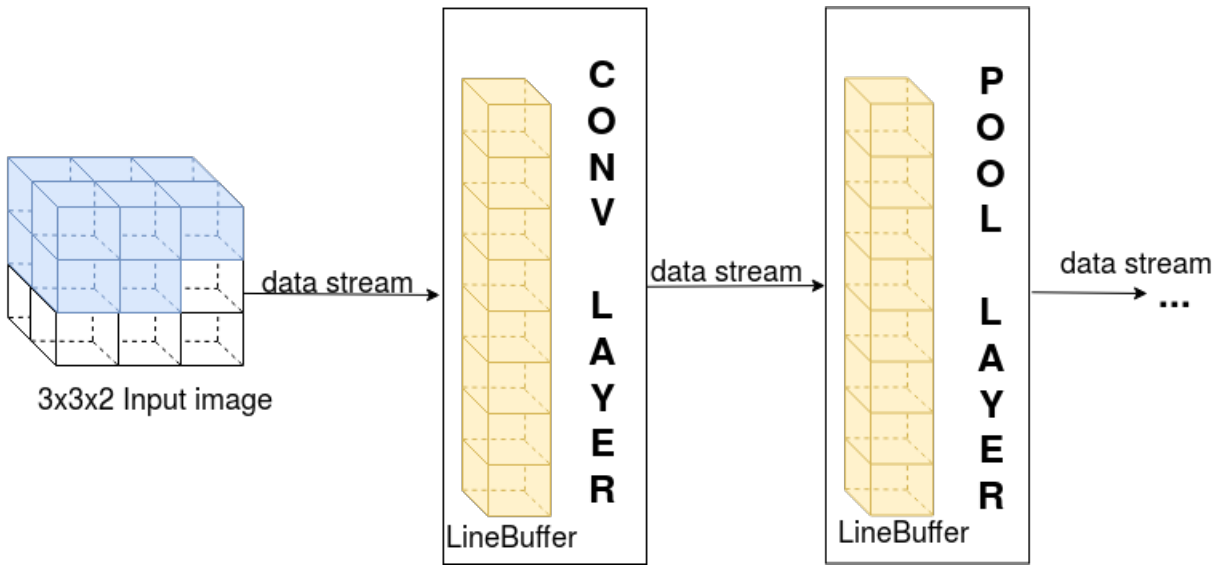


Figure 3: Dataflow modeling

The line buffers are dimensioned in such a way that they can hold enough data to process one output for the next layer. After the output is computed, one element exits the buffer while another one enters and is used to compute the next output. However, the last fully connected layer needs all the data from the previous layer to compute the output and hence, is executed sequentially.

Using float data type for representing image pixel values, weights and features gave same predictions as the trained model but resulted in significant increase in hardware. Fixed point notation with 5 bits for integer part and 11 bits for fractional part (total-16 bits) resulted in significant hardware savings with a minimal impact on prediction since the end goal is that of binary classification.

Vivado HLS contains various #pragma directives that can optimise the generated hardware to achieve desired latency and hardware consumption.

- #pragma HLS DATAFLOW is used to enable concurrent execution of convolution and pooling layers which is possible since data streams are used to pass data between functions and line buffers are used.
- #pragma HLS unroll is enabled by default that unrolls every loop to improve throughput.

The INTERFACE pragma specifies how RTL ports are created from the top function definition during interface synthesis. Following are the interface definitions for the top function-cnn:

```
#pragma HLS INTERFACE axis port=image_in
#pragma HLS INTERFACE axis port=out
#pragma HLS INTERFACE s_axilite port=return
```

The port=return attribute specifies that the function's return value will be connected to the AXI Lite interface and hence, can be used to start or stop cnn operation, check if our IP is busy, etc and all the other facilities provided by AXI4-Lite interface.

Minimum Latency achieved is 660848 clock cycles at a clock frequency of 100 MHz while keeping the FPGA resource utilization within limits.

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../cancerclassificationusingcnn_test.cpp in debug mode
   Compiling ../../../cancerclassificationusingcnn.cpp in debug mode
   Generating csim.exe
Prediction : 0.951172
Class : 1
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Figure 4: C Simulation Prediction Results for class 1 label

C Simulation was carried out for 20 images of both classes and the accuracy obtained was $\approx 80\%$.

Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
s_axi_AXILiteS_AWVALID	in	1	s_axi	AXILiteS	return void
s_axi_AXILiteS_AWREADY	out	1	s_axi	AXILiteS	return void
s_axi_AXILiteS_AWADDR	in	4	s_axi	AXILiteS	return void
s_axi_AXILiteS_WVALID	in	1	s_axi	AXILiteS	return void
s_axi_AXILiteS_WREADY	out	1	s_axi	AXILiteS	return void
s_axi_AXILiteS_WDATA	in	32	s_axi	AXILiteS	return void
s_axi_AXILiteS_WSTRB	in	4	s_axi	AXILiteS	return void
s_axi_AXILiteS_ARVALID	in	1	s_axi	AXILiteS	return void
s_axi_AXILiteS_ARREADY	out	1	s_axi	AXILiteS	return void
s_axi_AXILiteS_ARADDR	in	4	s_axi	AXILiteS	return void
s_axi_AXILiteS_RVALID	out	1	s_axi	AXILiteS	return void
s_axi_AXILiteS_RREADY	in	1	s_axi	AXILiteS	return void
s_axi_AXILiteS_RDATA	out	32	s_axi	AXILiteS	return void
s_axi_AXILiteS_RRESP	out	2	s_axi	AXILiteS	return void
s_axi_AXILiteS_BVALID	out	1	s_axi	AXILiteS	return void
s_axi_AXILiteS_BREADY	in	1	s_axi	AXILiteS	return void
s_axi_AXILiteS_BRESP	out	2	s_axi	AXILiteS	return void
ap_clk	in	1	ap_ctrl_hs	cnn	return value
ap_rst_n	in	1	ap_ctrl_hs	cnn	return value
interrupt	out	1	ap_ctrl_hs	cnn	return value
image_in_V_V_TDATA	in	16	axis	image_in_V_V	pointer
image_in_V_V_TVALID	in	1	axis	image_in_V_V	pointer
image_in_V_V_TREADY	out	1	axis	image_in_V_V	pointer
out_r_TDATA	out	16	axis	out_V_value_V	pointer
out_r_TLAST	out	1	axis	out_V_last	pointer
out_r_TVALID	out	1	axis	out_V_last	pointer
out_r_TREADY	in	1	axis	out_V_last	pointer

Figure 5: Custom CNN IP Interface Ports and Protocols

TLAST signal is added to a user defined structure used for out stream to connect it to S2MM interface of DMA to signal the DMA that the output stream has finished transmitting data.

Synthesis Report for 'cnn'

General Information

Date: Mon May 1 04:43:55 2023
Version: 2018.3 (Build 2405991 on Thu Dec 06 23:56:15 MST 2018)
Project: CancerClassificationUsingCNN
Solution: solution1
Product family: zynq
Target device: xc7z020clg400-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.719	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
660848	668960	660840	668952	dataflow

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2
FIFO	0	-	25	140
Instance	10	34	32143	19069
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	10	34	32168	19211
Available	280	220	106400	53200
Utilization (%)	3	15	30	36

Figure 6: Custom CNN IP Performance and Utilization Estimates

2.3 Block Design using Vivado

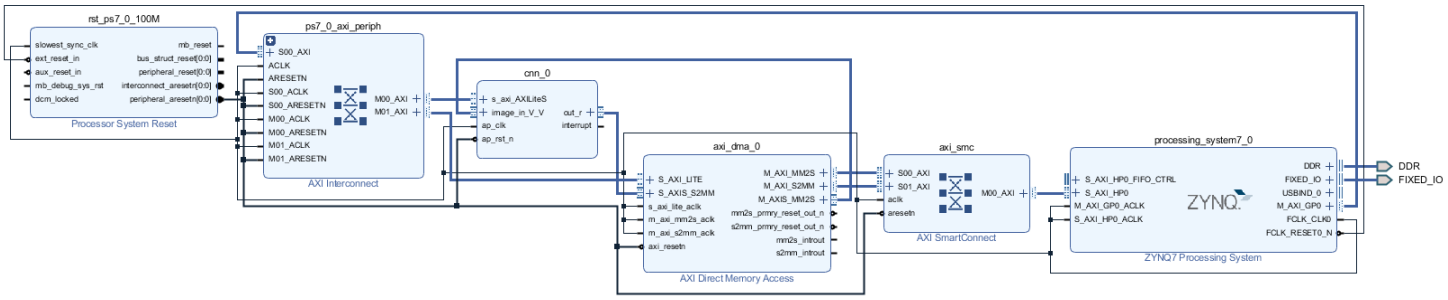


Figure 7: Custom IP Integration in Pynq

Zynq Processing system is used to configure DMA, our custom IP CNN and send it various commands to start, stop, check if busy/done, etc through GP0 (General Purpose) AXI4-Lite interface.

Similarly, data transfer between Zynq PS and DMA takes place through High Performance (HP0) AXI interface port. DMA is mapped to address range 0x10000000-0x1FFFFFFF with 32-bit addressing and has a stream width of 16 bits.

DMA pushes image array that it receives from PS (which in turn obtains it from DDR/Cache) to our custom IP CNN through M_AXIS_MM2S interface.

Once our IP performs inference, it passes prediction result to PS through DMA via S_AXIS_S2MM interface by asserting TLAST.

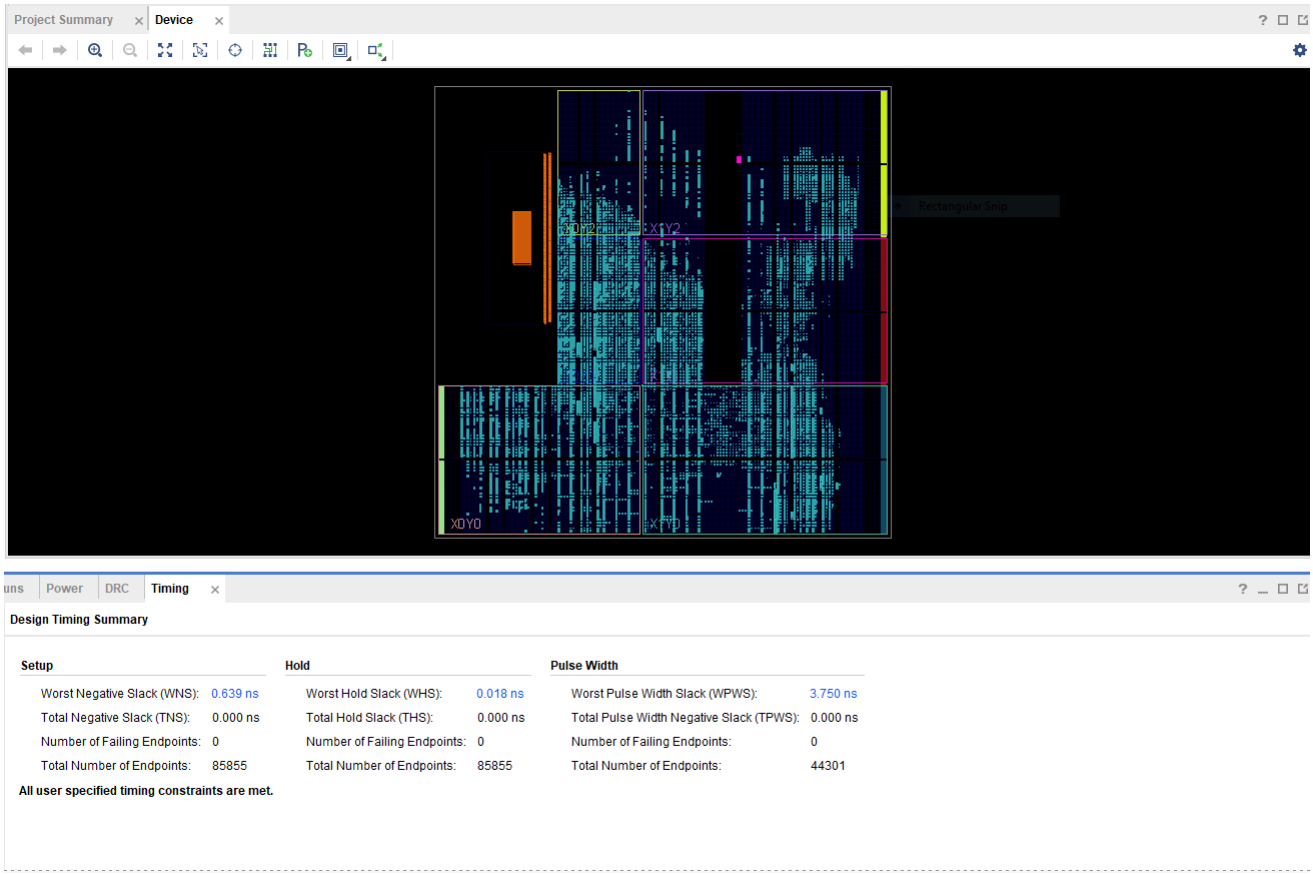


Figure 8: Implemented Design

2.4 Code Development using Xilinx SDK

Some of the important configurations are explained below:-

- Defining the addresses of DMA transmit and receiver buffers:

```
#define MEM_BASE_ADDR 0x01000000
#define TX_BUFFER_BASE (MEM_BASE_ADDR + 0x00100000)
#define RX_BUFFER_BASE (MEM_BASE_ADDR + 0x00F00000)
```

- Transmitting image_in_HW array to DMA and receiving data from DMA in m_dma_buffer_RX buffer:

```
XAxiDma_SimpleTransfer(&AxiDma, (UINTPTR)image_in_HW, IMAGE_SIZE*sizeof(float16_t),
                       XAXIDMA_DMA_TO_DEVICE);
XAxiDma_SimpleTransfer(&AxiDma, (UINTPTR)m_dma_buffer_RX, 1*sizeof(float16_t),
                       XAXIDMA_DEVICE_TO_DMA);
```

- Wait for DMA transfers to complete:

```
while (XAxiDma_Busy(&AxiDma, XAXIDMA_DMA_TO_DEVICE));
while (XAxiDma_Busy(&AxiDma, XAXIDMA_DEVICE_TO_DMA));
```

- Wait for convolution operation to complete:

```
while(!XCnn_IsDone(&cnn));
printf("Finished convolution \n");
printf("%f\n",(float16_t)m_dma_buffer_RX[0]);
```

3. Comparison with initial proposal

1. Initial proposed implementation was on TIVA-C microcontroller instead of Pynq FPGA.
2. Future improvement can include integration of Camera sensor through PMOD for real time cancer detection.

4. Components and Softwares Used

1. TUL PYNQ-Z2 Development Board
2. Jupyter Notebook - Model Training and Tuning
3. Vivado HLS 2018.3 - C Simulation and Synthesis
4. Vivado 2018.3 - Block Design and Bitstream Generation
5. Xilinx SDK - Application Development

5. References

1. https://github.com/basveeling/pcam
2. https://youtube.com/playlist?list=PLo7bVbJhQ6qzK6ELKCm8H_WEzzcr5YXHC
3. https://docs.xilinx.com/v/u/2018.3-English/ug871-vivado-high-level-synthesis-tutorial
4. https://docs.xilinx.com/v/u/2018.3-English/ug902-vivado-high-level-synthesis