Neural Network Accelerator

Toms Jiji Varghese, DESE, Indian Institute of Science, Bangalore-560012

I Students

1. Manish Kumar Singh - 23654
2. Toms Jiji Varghese - 22567
3. Narayanan Nampoothiry V - 23747
4. Sanidhya Saxena - 22586

II Introduction and Motivation

The rise of deep learning has fueled advancements in various fields, from medical diagnosis to autonomous vehicles. However, running these complex algorithms on resource-constrained devices presents a significant challenge. Traditional CPUs struggle with the massive amount of parallel computations required for neural network inference, leading to slow processing times. While GPUs offer superior performance, their high power consumption makes them less suitable for battery-powered devices like mobile phones and embedded systems.

FPGAs (Field-Programmable Gate Arrays) emerge as a compelling solution for accelerating neural networks on such devices. These versatile chips offer a unique blend of flexibility and performance. Unlike CPUs and GPUs with fixed architectures, FPGAs can be customized to efficiently execute specific neural network architectures. This allows for:

1. Parallel Processing: FPGAs contain a large number of configurable logic blocks that can perform computations simultaneously, significantly accelerating neural network inference compared to CPUs.
2. Hardware Customization: Unlike general-purpose processors, FPGAs can be tailored to the specific needs of the chosen neural network, leading to optimal resource utilization and potentially lower power consumption. This is particularly beneficial for resource-constrained devices.
3. Low Power Consumption: For specific tasks, FPGAs can achieve higher performance with lower power consumption compared to general-purpose processors due to their ability to leverage hardware-specific optimizations.
4. Cost-Effectiveness: Compared to Application-Specific Integrated Circuits (ASICs) FPGAs offer a significantly lower development cost. ASICs require significant upfront investment and are inflexible, while FPGAs can be reprogrammed for different neural network architectures as needed.

2.1 Objectives

The objective of this report is to implement the inference phase of a ANN on an FPGA efficiently. Concretely, the design shall have high parallelism, whereas the hardware resources used shall be minimized to reduce area and power consumption. The focus of this thesis will be reducing hardware resources and area. The runtime performance and power consumption of the design will be compared to that of a CPU and a GPU.

2.2 Hardware Advantage

A neural network accelerator FPGA (Field-Programmable Gate Array) is a specialized hardware device designed to efficiently execute neural network algorithms. FPGAs are programmable integrated circuits that can be configured after manufacturing, allowing for custom logic designs to be implemented. Several advantage that FPGA accelerator offers:

1. High Performance: FPGAs can be tailored to the specific requirements of neural network inference or training tasks, resulting in potentially higher performance compared to general-purpose processors or GPUs.
2. Flexibility: FPGAs can be reprogrammed or reconfigured to adapt to different neural network architectures or algorithms, providing flexibility in deployment.
3. Customization: FPGA designs can be customized to incorporate specific hardware features or optimizations tailored to neural network workloads, potentially leading to better performance or efficiency.

III Background Study

3.1 Neural Networks

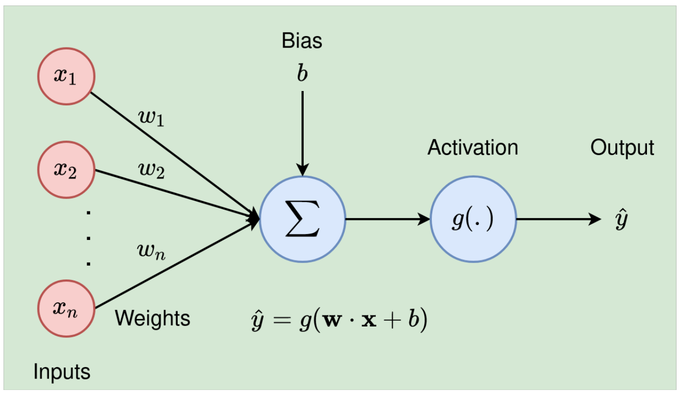


Figure 1: An aritificial neuron with n input values

Artificial Neural Networks (ANNs) are loosely inspired by the biological structure of the human brain. They consist of interconnected layers of processing units called neurons. Each neuron receives weighted inputs from other neurons, performs a simple mathematical operation (activation function), and outputs a signal to the next layer. By adjusting these weights through a training process, neural networks learn to recognize complex patterns in data.

There are various types of neural networks, but this project will likely focus on Convolutional Neural Networks (CNNs), a prevalent architecture widely used in image recognition and computer vision tasks. CNNs utilize specialized layers like convolutional layers and pooling layers to extract features from images efficiently. These layers involve a large number of mathematical operations, making them well-suited for hardware acceleration on FPGAs.

3.2 Convolutional Neural Networks (CNNs)

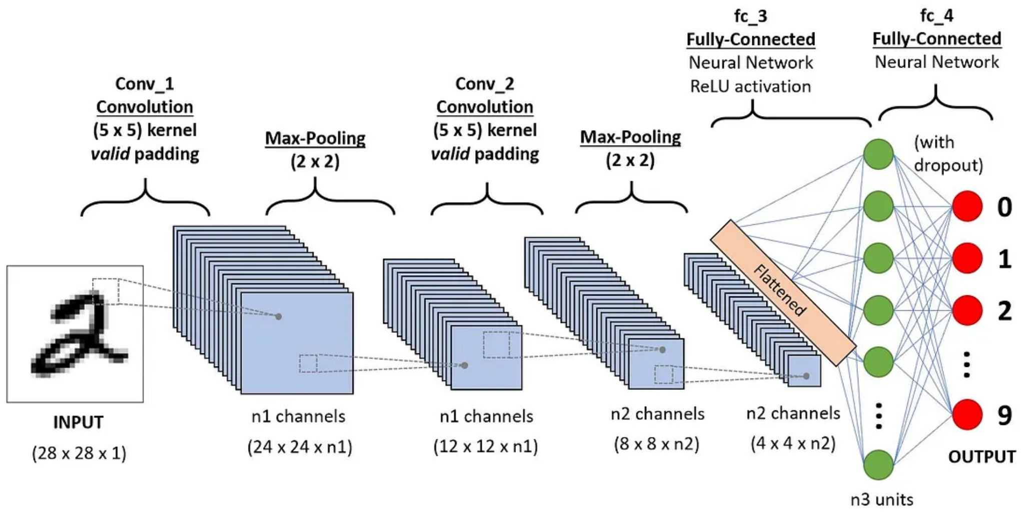


Figure 2: A CNN based digit recognition block diagram

Convolutional Neural Networks (CNNs) are a powerful type of artificial neural network architecture specifically designed for image recognition and computer vision tasks. Unlike standard neural networks that treat an image as a flat array of pixels, CNNs exploit the inherent spatial structure of images to achieve superior performance.

3.2.1 Convolutional Layers:

These are the building blocks of CNNs. They perform convolution, where a filter (also called a kernel) slides across the input image, extracting features like edges, lines, and shapes. Each filter learns to detect specific features at different locations in the image. By stacking multiple convolutional layers with varying filter sizes and orientations, CNNs can progressively capture increasingly complex features.

3.2.2 Pooling Layers:

These layers downsample the output of convolutional layers, reducing the dimensionality of the data and introducing some level of invariance to small shifts or rotations in the image. Common pooling techniques include max pooling, which selects the maximum value from a defined region in the feature map, and average pooling, which averages the values within that region. Pooling helps to control overfitting and reduces computational complexity.

3.2.3 Activation Functions:

Similar to standard neural networks, CNNs employ activation functions to introduce non-linearity into the network. This allows the network to learn complex relationships between the extracted features and the desired output. Popular activation functions used in CNNs include ReLU (Rectified Linear Unit) and Leaky ReLU, which help the network learn more discriminative features.

3.2.4 Fully-Connected Layers:

In the final stages of a CNN, fully-connected layers similar to those found in standard neural networks are used. These layers perform classification by taking the high-level features extracted by the convolutional and pooling layers and transforming them into class probabilities. The final output layer typically uses a softmax activation function to assign a probability score to each possible class.

IV Design Space Exploration and Design Strategies

4.0.1 Digit Recognition

Recognising handwritten numbers is a common computer vision challenge. Handwritten phone numbers, bank account numbers, postcodes, and other characters can all be recognised with it. As such, it is useful in real life. An image of digits contains much less information than many other images such as animals or human faces. On the other hand, there are only ten categories for digits, i.e., 0-9. These facts make it relatively easier to built a digits-recognition ANN on an FPGA

4.0.2 Artificial Neural Network

An artificial neural network (ANN) is a computational model inspired by the structure and functioning of biological neural networks, particularly the human brain. It consists of interconnected nodes, called neurons or units, organized in layers. Each neuron receives input signals, processes them, and then passes the output to the next layer of neurons. Through a process called training, ANN learns to recognize patterns and relationships in data, making it capable of tasks such as classification, regression, clustering, and pattern recognition.

Figure 3 shows a typical ANN. This ANN has three layers: an input layer, a hidden layer, and an output layer. The inputs for an ANN are called features, which represent important properties of the object to be classified or predicted

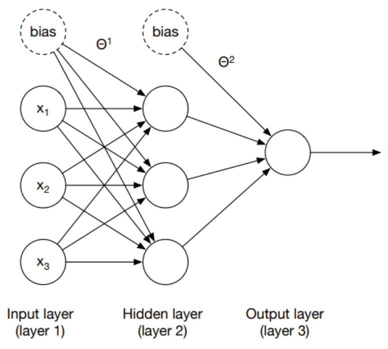


Figure 2.1: Typical ANN with three layers

Figure 3: ANN

4.0.3 Activation Function

In an ANN, an activation refers to the value or the output of a node . It is calculated by applying an activation function on the summation of the products of the inputs of the node and corresponding weights . Activation functions are a crucial component of artificial neural networks (ANNs) as they introduce non-linearity into the network, allowing it to learn complex patterns and relationships in data. Without activation functions, ANNs would essentially reduce to a series of linear transformations, severely limiting their expressive power.

4.0.4 Sigmoid Activation function

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

- Produces values between 0 and 1.
- However, it is compute intensive.

4.0.5 Rectified Linear Unit (ReLU):

$$ReLu(x) = max(0, x)$$

- Simple and computationally efficient.
- Overcomes vanishing gradient problem for positive inputs.
- However, it suffers from the "dying ReLU" problem where neurons can become inactive for negative inputs during training.

4.0.6 Forward Propagation and Backward Propagation

In an ANN, forward propagation simply refers to passing the inputs forward through each node to compute the final output. Analogously, backward propagation refers to passing values backwards to calculate the gradients of the loss function and update the weights in the network. Backward propagation is only used in the training phase. After many iterations, the error of the network is reduced and the accuracy is improved. This process starts with computing the error between the network's output and the correct value. Then it goes to the previous layer until reaches the first layer of the network.

V IMPLEMENTATION

ANN implementation contains an Input layer which is fed the image pixels. Size of the input layer is the size of an image (28X28), Hence 784 input pixels. Then these pixels are fed the to the hidden layer and finally output is generated between 0 to 9.

Figure 4 below shows the block digaram of our implementation. We have the input pixels stored in memory prior to implementation.

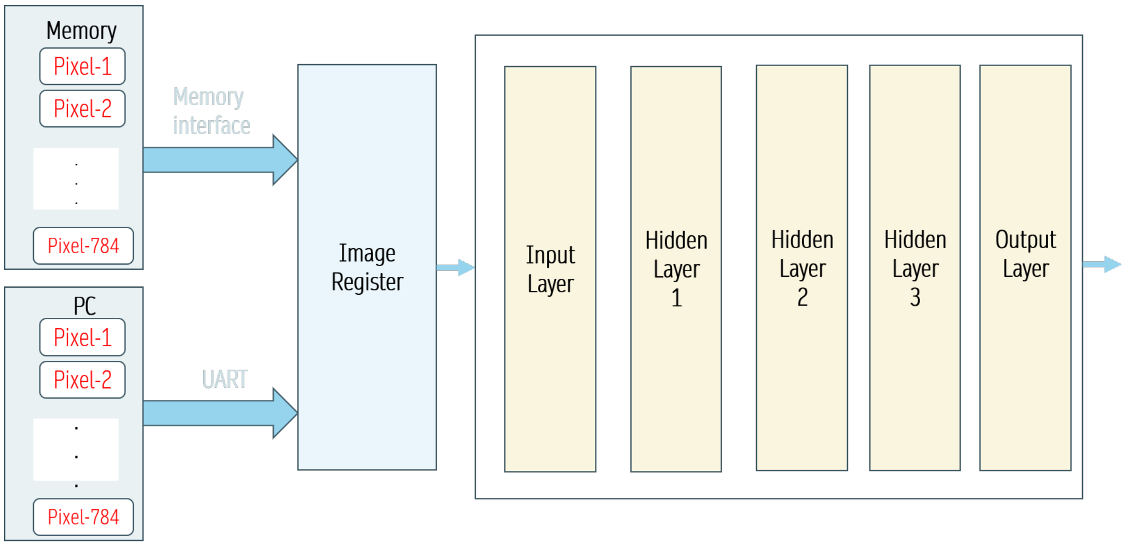


Figure 4: ANN Implementation block diagram

In this implementation, we can have 2 modes of operation. Either the input pixels can be stored in memory prior to the implementation, or we can give the image pixels in the real time using UART communication.

Hidden layers play a crucial role in the representational power and learning capacity of neural networks, allowing them to model complex relationships in data and solve a wide range of machine learning tasks effectively.

5.1 *MicroArchitecture*

Figure 5 below shows the micro architecture of our implemented design.

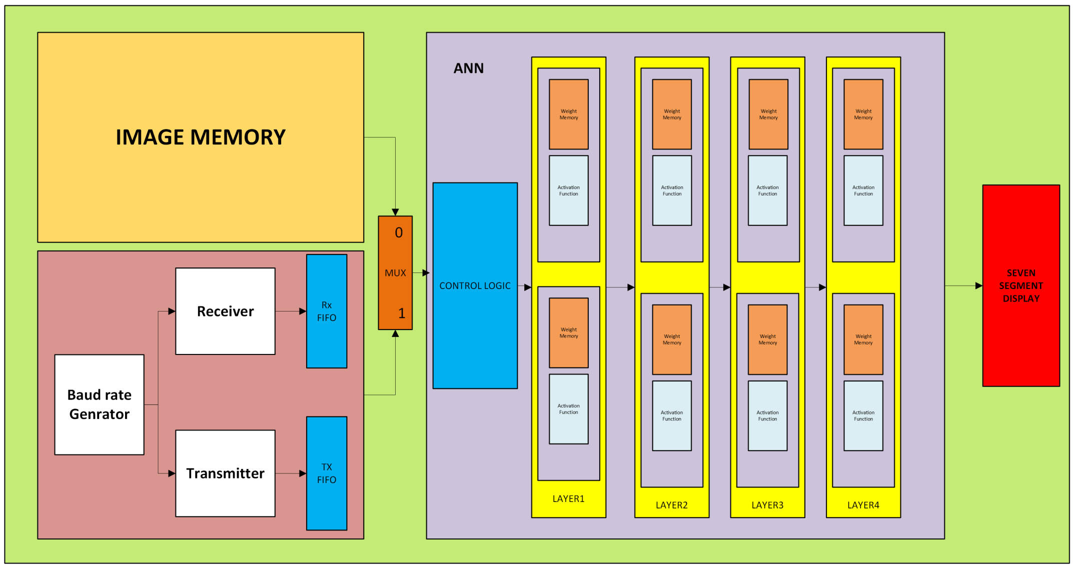


Figure 5: Microarchitecture ANN

In our design we have 3 major subsystem,

1. Memory Subsystem: BRAM is instantiated which stores the image pixel pre-hand before the implementation.
2. UART Subsystem: We have designed the UART Rx and Tx driver which takes the image at certain baudrate from PC using COM ports and the output is then fed to the ANN module
3. ANN Subsystem: ANN subsystem is the ANN implementation part which has 3 hidden layer and 1 output layer.

Finally, the result is displayed on seven segment display.

5.2 DESIGN SCHEMATIC

Figure 6 shows the DNN Top schematic. It instantiates UART Driver, Image BRAM and ANN top as shown in the microarchitecture.

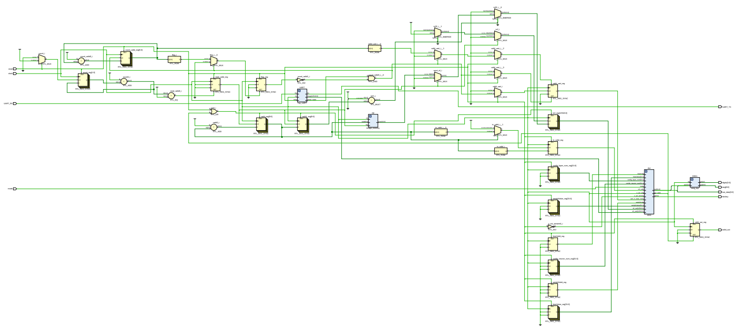


Figure 6: DNN Top Schematic

5.3 UART Subsystem

UART (Universal Asynchronous Receiver/Transmitter) modules are commonly implemented in Field-Programmable Gate Arrays (FPGAs) to enable serial communication between the FPGA and other devices. Here's an overview of how UART modules are typically implemented in FPGAs.

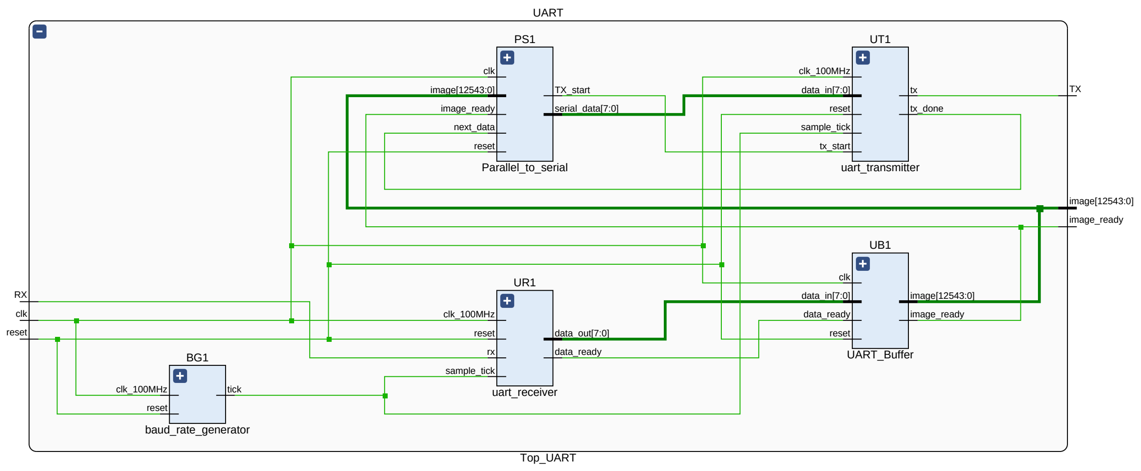


Figure 7: UART Schematic

1. Baud rate generator: It produces a tick at every instance to create a final value such that every bit is transmitted at a given baud rate. For every but, we'll oversample and then will find the bit sequence as per UART protocol.

$$FinalValue = \frac{1}{16 * b * T} - 1$$

   Where b is baud-rate, in this design is 9600 and T=10ns for 100mhz Fpga clock.

2. UART Receiver: It receives data bit and bit serially and then create a byte and send it to UART buffer.
3. UART Buffer: This is like a fifo, it gets each byte from the receiver module and when it gets 1568 Bytes, then it sends a ready signal to ANN module to sample the data.

4. Parallel-To-Serial-Converter: This block is used during the loopback. 1568 bytes from the buffer goes to this module and it converts that data into a stream of bytes. These Bytes then go to transmitter module for the transmission back to PC

5. Transmitter Module: Just like receiver, this blocks works on UART protocol. It gets a byte data from P2S block and generate a stream of bits to send back to PC. Start bit, followed by 8-bit data starting from D0-D7 and then stop bit.

5.4 Convolution Block

A convolutional layer extracts the important features of an image such as vertical edges and horizontal edges by performing convolution. In image processing, a convolution is defined as

$$y[i,j] = \sum \sum f[m,n]x[i+m,j+n]$$

where y[i, j] is the result after convolution, m and n are the width and height of the convolution kernel f , and x[i, j] is the original pixel value.

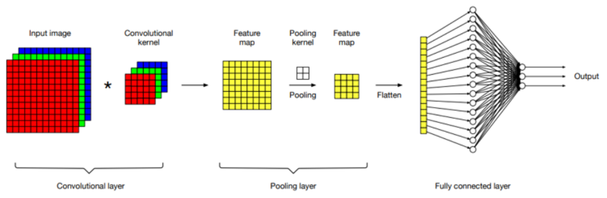


Figure 8: Image Convolution

The above image 8 shows how the convolution has to be done on same image with multiple kerners, so as to extract different feature each time we convolute.

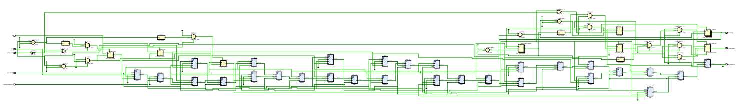


Figure 9: Convolution Block Schematic

Figure 9 shows the convolution block schematic.

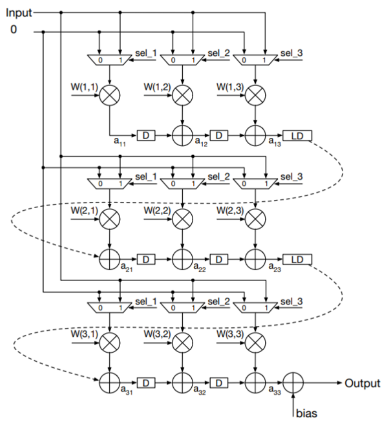


Figure 10: Convolution Block

The circuit shown in figure 10 is designed for a 3 ˆ 3 convolution window. The number of rows and the number of adders and multipliers in each row are determined by the kernel size. One pixel is read every clock cycle and multiplied by all the coefficients simultaneously. Partial sums are stored in buffers denoted by D, which hold the value for one clock cycle, and LD, which holds the value for several clock cycles because we have to skip those values that are not needed for calculating the result of the current convolution window. The length of a LD buffer is computed as m - f , where m is the image size and f is the convolutional kernel size. The last adder is used for adding bias. One output is produced every clock cycle. This structure is simple and easy to implement. However, it is inefficient to use this structure to handle image padding. Coefficients need to be changed to zero to support image padding which means more memory accesses are required

5.5 *Max Pooling*

A convolutional layer is usually followed by a pooling layer that reduces the spatial dimension of feature maps. A pooling layer works similarly to a convolution layer. The difference is that convolution kernels are replaced by pooling kernels. Max Pooling and Average Pooling are two commonly used pooling algorithms. Max pooling is defined as

$$y_i = Max(x_j), j\epsilon[1, m^2]$$

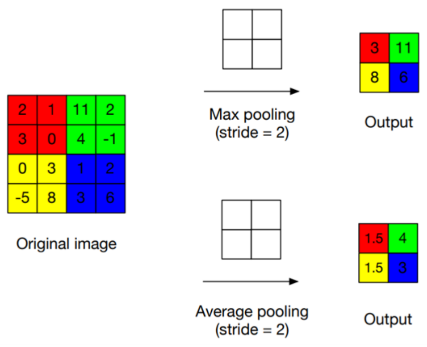


Figure 11: Max Pooling

Figure 12 shows the RTL schematic of max Pooling Block.

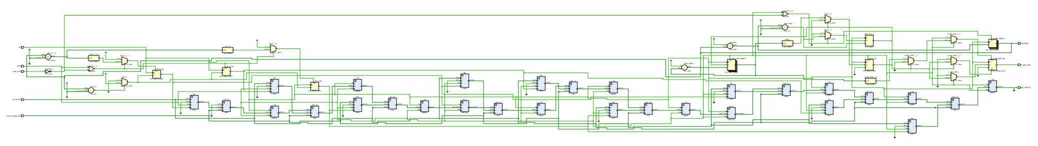


Figure 12: Max Pooling RTL Schematic

Pooling layer S2 uses 2ˆ2 pooling kernels. Figure 13 shows the schematic of an pooling kernel.

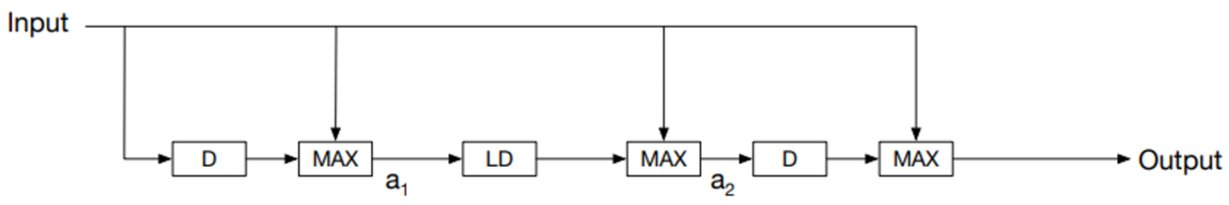


Figure 13: Max Pooling Design

One pixel is read by an S2 kernel every clock cycle. The first input is stored in a buffer denoted by D and delayed by one clock cycle, the next clock cycle the second input is compared with the first one that stored in the buffer. The larger value is stored in a long buffer denoted by LD, the length of LD is computed as m ´ f + 1 where m is the image size and f is the pooling kernel size. Then, the third input is compared with the partial result stored in LD, the larger value is again stored in a buffer and to be compared with the last input. The final output is the maximum value in the pooling window. It takes m2 clock cycles for an S2 kernel to finish one feature map, where m is the size of the input feature map.

5.6 ANN Module

The design is done in a modular way using neurons generating layers and layers forming the total ANN. Figure 14 shows the ANN RTL schematic.

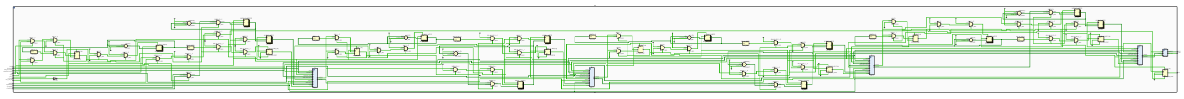


Figure 14: ANN Schematic

The architecture of a single artificial neuron used by ANN is as shown in Fig 15 below.

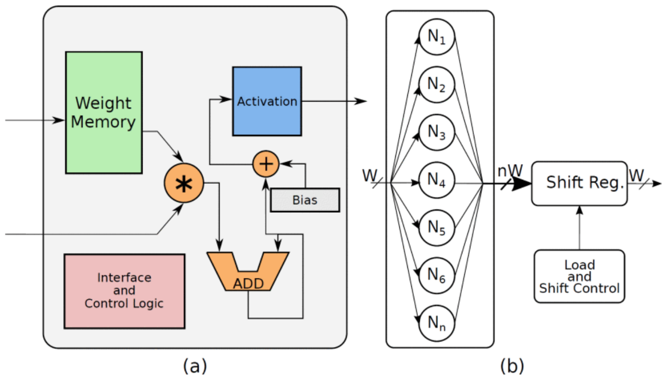


Figure 15: Neuron implementation

Each neuron has independent interfaces for configuration (weight, bias etc.) and data. Irrespective of number of predecessors, each neuron has a single interface for accepting data. This enables scalability of the network and improves clock performance by compromising on latency.

An internal memory, whose size is decided by the number of inputs to the neuron, is used to store the weight values corresponding to each input. Depending on whether the network is configured as pre-trained or not, either a RAM with read and write interface or a ROM initialized with weight values are instantiated. As inputs are streamed into the neuron, a control logic reads the corresponding weight value from the memory. Inputs and corresponding weight values are multiplied and accumulated (MAC) and finally added with the bias value. Like weights, bias values are stored in registers at implementation time if the network is pre-trained or configured at runtime from software. The output from the MAC unit is finally applied to the activation unit. Based on the type of activation function configured (Sigmoid, ReLU, hardMax etc.), either a look-up-table based (for Sigmoid) or a circuit-based function is implemented by the tool. The type of function chosen has a direct impact on the accuracy of the network and the total resource utilization and clock performance. The depth of the LUT for Sigmoid function can be optionally specified by the user or the tool can automatically determine it.

Each layer instantiates user specified number of neurons and manages data movement between the layers. Since each neuron has a single data interface and a fully connected layer requires connection to every neuron from the previous layer, data from each layer is initially stored in a shift register. It is then shifted to the next layer one per clock cycle. Connection between layers and integration with input and output AXI interfaces are automatically implemented by the tool.

Due to the modular layered structure of the implementation, correct signalling between the modules are a necessity for the accurate results. So, various signals were generated and passed between the modules for functioning. Also, where ever possible a level of standardization is followed. For eg. data bus width throughout the ANN module is standardized to 8 bits and the register size is 16 bits.

Initially, UART module will be copy the data to the Layer0 register and signals that the data is ready for processing. Layer1 takes this 784*16 bit long data as pieces of 8 bits and do the processing. Further after the first layer, layer1 will signal the layer2 that the data is ready for the next layer processing. In this manner the data will pass through the 4 layers and reaches the 10 output nodes. A HardMax module is used to normalize and report the most-likely input digit.

VI Implementation Challenges

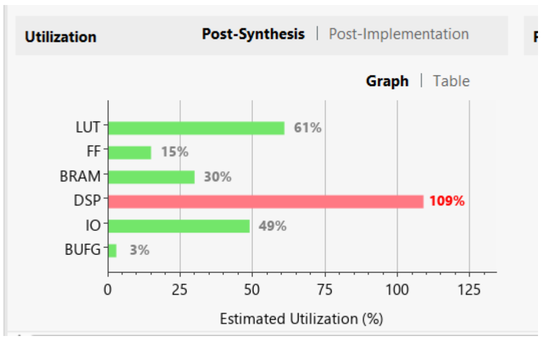


Figure 16: Lack of DSP slices post synthesis on BASYS 3 board

During the implementation the following challenges were faced:

- **Limited DSP Slices:** The BASYS 3 board only offered 98 Digital Signal Processing (DSP) slices, which were insufficient for our needs. To address this, we switched to the NEXYS A7 board, which provided 160 DSP slices.
- **Endianness Issues with Image Data:** We encountered difficulties transmitting the image data, which consisted of 16-bit pixels, through the Universal Asynchronous Receiver/Transmitter (UART). Initially, the data was sent in big-endian format (where the most significant byte comes first), causing issues. We had to ensure the data was transmitted in the correct order and endianness which was verified by implemting a custom UART transmitter which is send back the entire received image to the MATLAB script for verification.
- **UART State Machine Bug:** During the design of the UART state machine, which assembles received data into an image, we faced a discrepancy. The behavioral simulation (a functional simulation) worked as expected, but the post-synthesis simulation did not. This was traced back to a bug in the code that prevented the entire "always block" (a core element in FPGA design) from being synthesized.

VII Simulation Results

7.1 Convolution Block Verification

Figure 17 below shows the waveform which takes image matrix as the input and kernel matrix and produces a comvolved matrix.

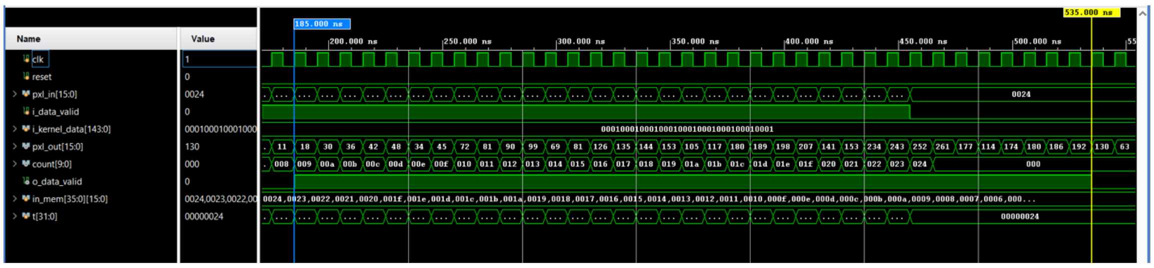


Figure 17: Convolution Waveforms

$$\begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 \\ 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 & 36 \end{matrix} * \begin{matrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{matrix} \rightarrow \begin{matrix} 18 & 30 & 36 & 42 & 48 & 34 \\ 45 & 72 & 81 & 90 & 99 & 69 \\ 81 & 126 & 135 & 144 & 153 & 105 \\ 117 & 180 & 189 & 198 & 207 & 141 \\ 153 & 234 & 243 & 252 & 261 & 277 \\ 114 & 174 & 180 & 186 & 192 & 130 \end{matrix}$$

This is the input image pixel (6X6), convoluted with a (3X3) kernel and produces a result of (6X6) due to edge correction. These outputs starting from 18 can be seen in the waveform and the outvalid signal goes high. This matrix is then fed serially to the max-pooling layers.

7.2 *Max Pooling Verification*

Figure 18 below shows the waveform which takes convoluted matrix as the input and produces a max-pooled output matrix.

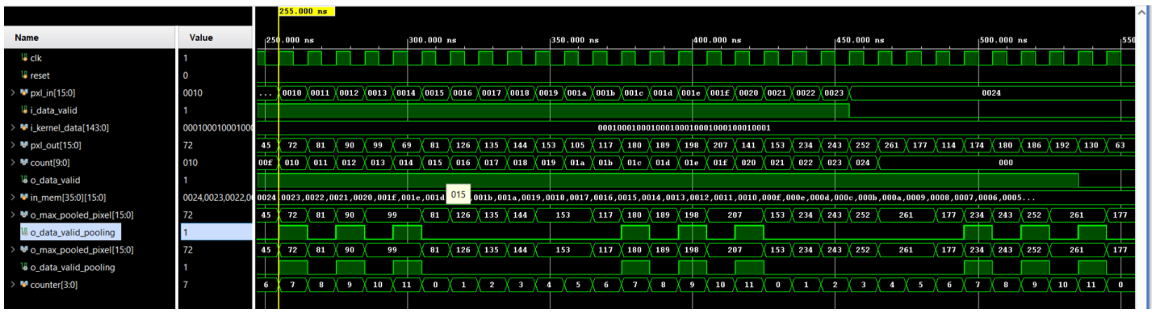


Figure 18: Max Pooling Waveform

$$\begin{matrix} 18 & 30 & 36 & 42 & 48 & 34 \\ 45 & 72 & 81 & 90 & 99 & 69 \\ 81 & 126 & 135 & 144 & 153 & 105 \\ 117 & 180 & 189 & 198 & 207 & 141 \\ 153 & 234 & 243 & 252 & 261 & 277 \\ 114 & 174 & 180 & 186 & 192 & 130 \end{matrix} \quad ------> \quad \begin{matrix} 72 & 90 & 99 \\ 180 & 198 & 207 \\ 234 & 243 & 261 \end{matrix}$$

We can see in the waveform that it produces the output 72 onwards, but since outputs are not continuos we need to store it in fifo for the next stage to process it.

7.3 ANN Verification

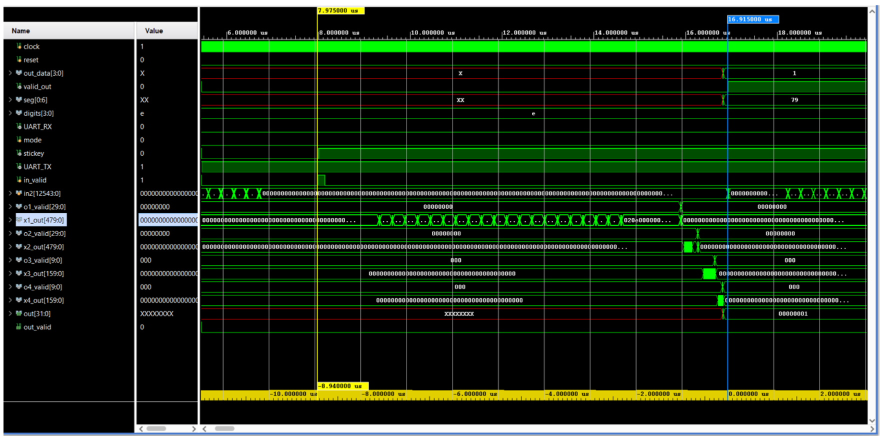


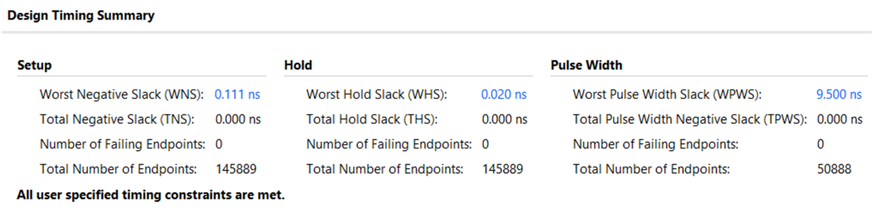
Figure 19: ANN Waveform

For the ANN we had taken a self-checking testbench. It takes the input from the image_001.txt file, which contains the 784 pixels of test data image. These inputs are then fed to the ANN module parallely, emulating a memory. In the above image, when the in_valid goes high, the data is sampled by the ANN module. Now as the input goes into the hidden layer, at each layer we get an output and output_valid. After the 4 layers and hard-max, we get the result from 0-9 which represent the image number we gave. In this testcase, we gave an image number as 1 and we get the expected output as 1 only.

VIII Implementation Results

8.1 Timing Check

The design is running on 100Mhz, and we get a positive timing slack. Hence the timings are met for our design.



Design Timing Summary

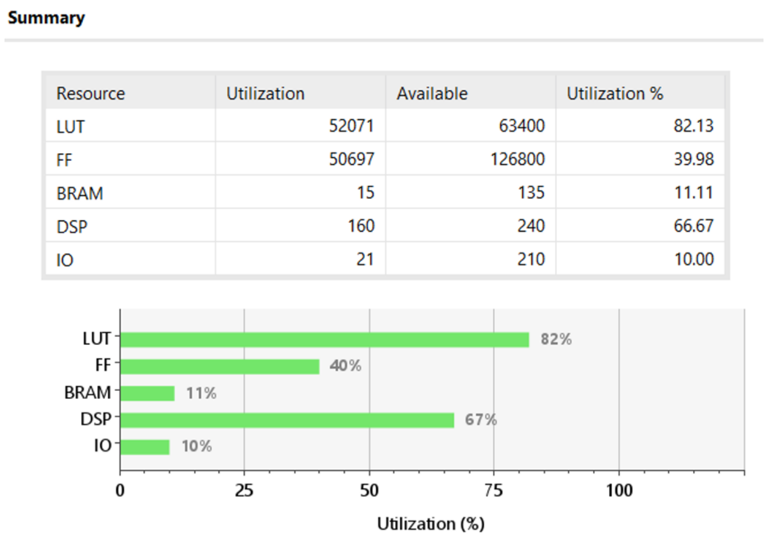
Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.111 ns	Worst Hold Slack (WHS):	0.020 ns	Worst Pulse Width Slack (WPWS):	9.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	145889	Total Number of Endpoints:	145889	Total Number of Endpoints:	50888

All user specified timing constraints are met.

Figure 20: Timing Anaysis

8.2 Resourse Utilisation

Resourse Utilisation is showed below in the figure 21,



Summary

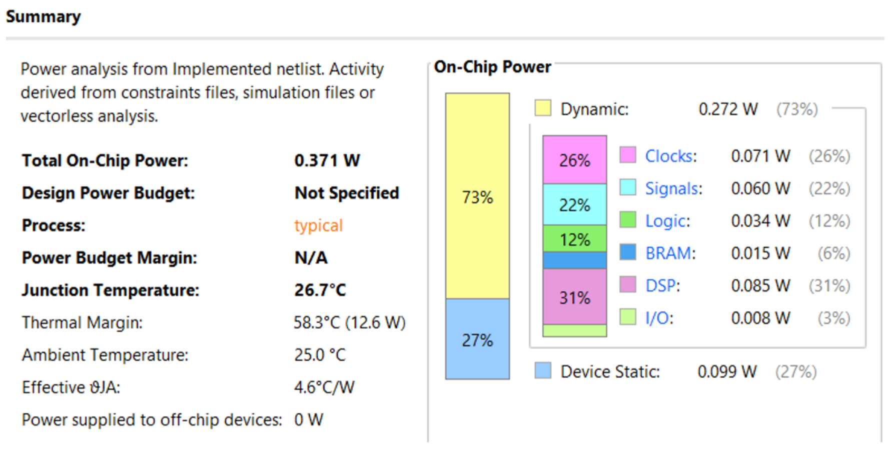
Resource	Utilization	Available	Utilization %
LUT	52071	63400	82.13
FF	50697	126800	39.98
BRAM	15	135	11.11
DSP	160	240	66.67
IO	21	210	10.00

Figure 21: Resource Utilisation

Image Pixels of size 784*16 bits i.e 12.5KB of memory is used for this. Most of the DSP is utilised in the ANN design due to the presense of perceptron module, which contains the MAC unit.

8.3 Power Budget

Figure 22 below shows the power budget for our design.



Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.371 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**26.7°C**
Thermal Margin:	58.3°C (12.6 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W

On-Chip Power

Dynamic:		0.272 W	(73%)
	Clocks:	0.071 W	(26%)
	Signals:	0.060 W	(22%)
	Logic:	0.034 W	(12%)
	BRAM:	0.015 W	(6%)
	DSP:	0.085 W	(31%)
	I/O:	0.008 W	(3%)
Device Static:		0.099 W	(27%)

Figure 22: Power Budget

Out of the total power, 73% is the dynamic power. Majority of the power is consumed by the DSP blocks and the CLOCK module. Hence we can place some ICG cells to gate the clock of modules which were not in used during the stages of the computations.

IX Post Implementation Testing

9.1 Custom MATLAB Script for Testing:

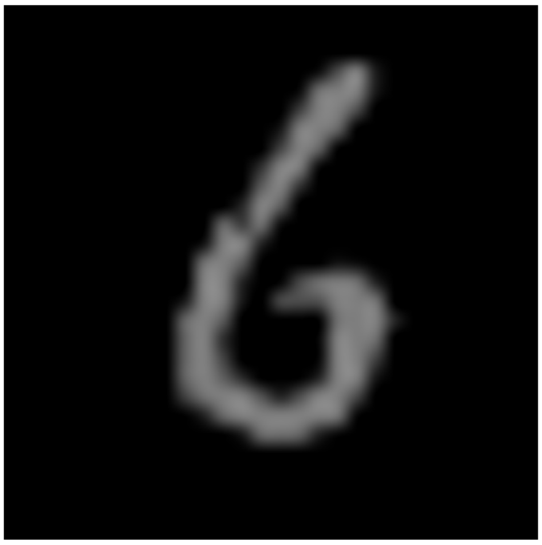


Figure 23: MATLAB output image

To test the implemented data transmission system, we developed a custom MATLAB script. This script performs the following functions:

- **Input Handling:** It takes an MNIST image in PNG format as input. MNIST is a common dataset for handwritten digit recognition.
- **Data Processing:** The script breaks down each 16-bit pixel value in the image into two separate 8-bit chunks.
- **Data Transmission and Visualization:** It then transmits these 8-bit chunks via the UART console. Simultaneously, the script displays the original image on the screen for reference. This allows us to visually verify that the transmitted data corresponds to the original image.

9.2 Python Script for Testing:

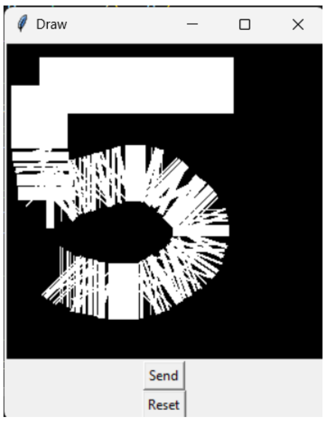


Figure 24: Python image input console

A custom Python script was also developed to test user interaction with the system. This script allowed users to input characters via the UART console as shown in fig 24 , simulating real-time data transmission. As the neural network was not trained on a broader range of characters, the script's results confirmed the network's sensitivity to the specific dataset it was trained on (MNIST in this case). This demonstrates the importance of training neural networks with datasets that reflect the intended use cases.

X Conclusion

In conclusion, this report details the successful design and implementation of an FPGA-based neural network accelerator for digit recognition. The project addressed challenges like limited hardware resources, data transmission issues, and code bugs. While a fully-connected network was initially envisioned, the design incorporated convolutional and max pooling layers as an exploration for potential performance improvements. With the ANN implementation we got an accuracy of 98%. Post-implementation testing verified functionality, and resource analysis highlights opportunities for optimization. This project demonstrates the feasibility of utilizing FPGAs for digit recognition with neural network accelerators. Future work can explore integrating these additional layers into a complete CNN architecture, optimizing resource usage and power consumption, and investigating more complex network designs for broader applications.

References

[1] The Basics of Neural Networks (https://towardsdatascience.com/the-basics-of-neural-networks-neural-network-series-part-1-4419e343b2b)

[2] https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[3] https://www.3blue1brown.com/topics/neural-networks

[4] https://liu.diva-portal.org/smash/get/diva2:1367984/FULLTEXT01.pdf

[5] https://nafizshahriar.medium.com/what-is-convolutional-neural-network-cnn-deep-learning-b3921bdd82d5

[6] Handwritten Digit Classification on FPGA - Kais Kudrolli, Sohil Shah, DongJoon Park

[7] K. Vipin, "ANN: Automating Deep Neural Network Implementation on Low-Cost Reconfigurable Edge Computing Platforms," 2019 International Conference on Field-Programmable Technology (ICFPT), Tianjin, China, 2019, pp. 323-326, doi: 10.1109/ICFPT47387.2019.00058. keywords: {neural networks;hardware-software co-design},