Smart Automation Controller for Precision Agriculture

Design Project

Tarek Omar - 153143

Laila Hatem - 152493

Abdalla Mohamed - 164630

Ahmed Essam Rady - 129625

Abdelrahman Mohammad - 157900

Dr. Ihab Adly - Supervisor

Faculty of Engineering

Electrical Engineering Department

(Computer Engineering Programme)

Table of Contents

List of Figures

List of Tables

1. Introduction

The significant rise in the deep neural networks in image detection and classification applications created an emergent industry that is led by applicational innovation, such applications like medical diagnosis, market analysis, brain-computer interfaces and in many more emergent applications. This project focuses on off the grid agricultural farms, where the need to detect plant diseases through plant medical centers is limited. Therefore this project aims to create a power efficient and responsive edge device that can classify different plant diseases. These types of advancements in agricultural technology can reduce the application of excess pesticide use and provide precise application of antibacterial and antiviral antidotes, this directly improves the health of the plant and reflects onto our physical and mental aptitudes. Neural networks, especially Convolutional Neural Networks (CNNs), can provide the center point for such implementations. CNNs are prominent in image recognition and classification. With the advancements of modern toolkits like TensorFlow and others, the limit is the creativity of the engineer.

In the review section, this report covers the fundamental building blocks that lead to the creation of CNN architectures. It also covers how Field Programmable Gate Arrays (FPGAs) are the platform of choice for efficient and responsive systems that edge devices require. FPGAs are resource limited devices therefore, lightweight CNN architectures like MobileNet will be evaluated based on how they compare with more standard architectures. Lastly, methods of applying CNN models onto FPGAs are closely examined.

Moreover, a fully VHDL implementation of the MNIST dataset was developed. In the MNIST VHDL Implementation section, we discuss the major layers and how they were conjoined to form an overall system. By using functional, testbench and timing simulations, we can pinpoint and understand the low level implications of creating such a system in base the RTL flow.

Furthermore, the Applicational Implementation section will discuss our methodology in order to meet the requirements of the application, the many challenges that were met and the unique approaches generated to overcome them, in the same order that occurred during our implementation phase. Selecting CNN architectures that maintain a high accuracy threshold while also leaving a small storage footprint required research into relatively modern CNN architectures that are capable of being lightweight. After selecting appropriate architectures, training the model introduced many challenges that are documented in turn. Due to the dynamic nature of this project, we had to develop a low level custom CNN architecture that supports MobileNet V1 on Python, without the aid of TensorFlow libraries. This was accomplished by reviewing how TensorFlow implements their optimized functionalities and reconstructing them, with our approaches documented in turn. A full system functional simulation was developed in Python and through the use of pointer arrays and threading libraries we were able to emulate a shared memory component and simulate realistic real time components running and interfacing concurrently. By following industry techniques, we developed a custom single purpose processor for applying a matrix multiplication algorithm to VHDL. Given its inputs and a start signal, the RTL driven algorithm would run synthesized on the FPGA. Exploring an interfacing medium, we researched Avalon Memory Mapped Interfaces, this allowed for a straightforward memory mapped peripheral design. In order to utilize the memory mapped medium we developed a custom peripheral component that wraps the single purpose processor and exposes the peripheral (Slave) to the interface medium for communication of the HPS and FPGA (Masters). Moreover, we developed a datasheet for the peripheral that documents the usability of the component. And lastly, joining all the designed components into the system implementation, this introduced the applications' limitations into clear view. We overcome the limitations and apply a base implementation, an iteration upon it and discuss their limitations and compare their results, we also introduce future work iterations that could provide a balance point between our two iterations.

2. Review

In order to attempt a novel design or implementation, research into CNN architecture and how its low level components interact was undertaken. Moreover, the available platforms that can efficiently utilize their resources into housing CNN models are reviewed. Furthermore, model architectures were reviewed in order to extract the most efficient imprint on FPGA with the least resource utilization. But firstly, a model implementation of a lightweight architecture (MobileNet) and what is considered the base architecture (AlexNet/ImageNet) were undertaken to compare, contrast and objectively ascertain which is most appropriate for this implementation. Moreover, approaches to apply the CNN models onto FPGAs were examined, HLS was looked into as a stepping stone implementation for faster iteration and better interface understanding. RTL's superior utilization and efficiency improvements over HLS pushed research and an initial implementation into CNNs on RTL.

2.1. CNN Architecture

Understanding the fundamentals of CNNs allows for low level understanding of the underlying operations that occur.

CNN's architecture is similar in theory across all implementations, a top level review into CNN models describes a 3 tier model of input, output and hidden layers; they hold intermediate feature maps extracted from the input image. CNNs are a subset of deep neural networks, they approach image recognition by extracting features from the input image via the use of kernels. Unlike in computer vision approaches these kernels are not predefined; they are however initially random.

During the model training phase, the CNN models carry out the forward propagation where calculations starting from the input layer are passed to the next, where various operations take place, until the model creates a prediction of the input. Based on the accuracy of the prediction the model's kernels have their weights adjusted via the use of the backpropagation algorithm and a supervised learning approach [19]; where based on error values of the prediction, the model manipulates the kernel values (weights) into having a more accurate future response. This occurs over multiple epochs until the engineer is satisfied with the fitness of their model. [17]

As shown, the hidden layer enclosed in red, typically consists of connected blocks that filter through the input in order to allow for feature extraction and then classification. The operations executed inside the hidden layer are executed in a loop, multiple times over. In practice, these three operations are set together and labeled as a Convolutional Block.

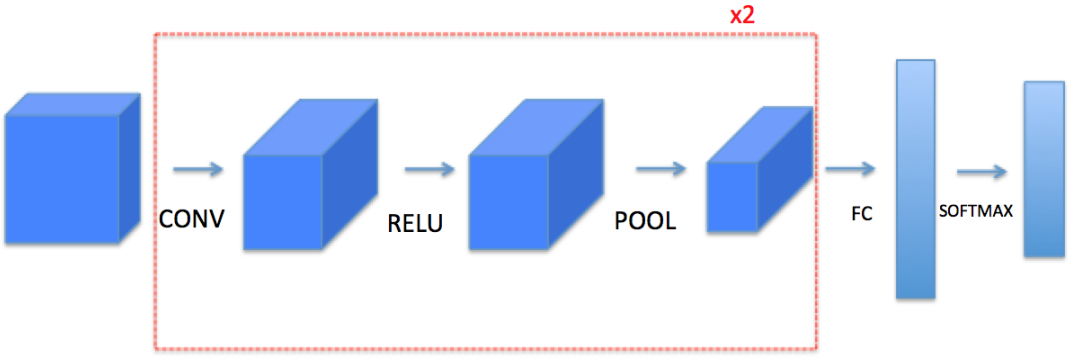


Figure 1 - CNN Structure [14]

2.1.1. Input Layer

Input layers of CNNs are concerned with acquiring the pixel values of the input image; in three dimensions (RGB). Which allows for the creation of the input of the next layer which is typically the convolutional layer. A preprocessing of inputs is required in order to preallocate the dimensionality of the tensor, which can be calculated via:
(#images in dataset) x (image dimensions). Where image pixel values populate the tensor.

2.1.2 Convolutional Layer

The convolutional layer consists of blocks that calculate the convolution operation of their input with the kernel. The kernel tensor dimensions are calculated via:
(kernel dimensions) x (image depth) x (#kernels) with the weights populating.

The convolution operation is executed by performing dot products; which is executed by multiplying input and kernel values element-wise, followed by a summation then adding the bias, if present, as shown in the below figure.

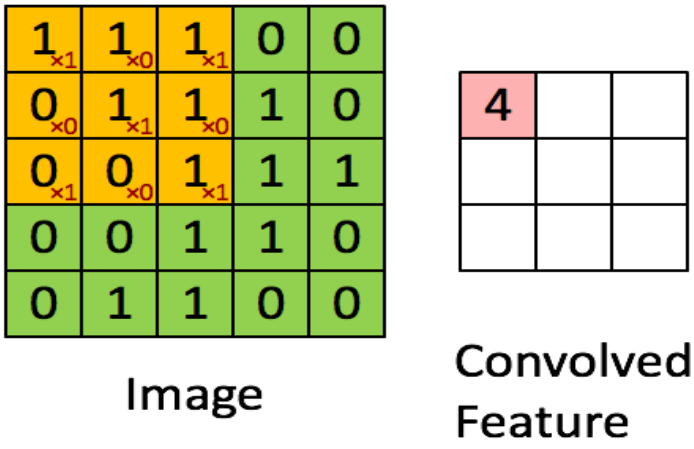


Figure 2 - Convolution Operation (Green: Input, Yellow: Kernel, Red: Output) [14]

2.1.2.1 Convolutional Hyper Parameters

Kernel sizes and numbers are variable depending on the architecture of the model. The convolution operation outputs requires dimensions of size (#images) x (n_H) x (n_W) x (#kernels) where n_H and n_W are preprocessed values obtained by using the equation shown below.

$$n_H = \lfloor \frac{n_{H_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$
$$n_W = \lfloor \frac{n_{W_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$
$$n_C = \text{number of filters used in the convolution}$$

Figure 3 - Preprocessing of Convolution Output Dimensions [14]

Convolution is supported by a set of hyperparameters which alter its operation. As the number of used kernels increases a greater feature bandwidth is allowed at the expense of performance and memory utilization. How far the filter will slide on each iteration of the convolution is determined by the stride parameter.

Lastly, the padding is used to preserve the output of the convolution, to avoid input dimensionality shrinkage. Shown in the below figure is the output of a padded convolutional operation.

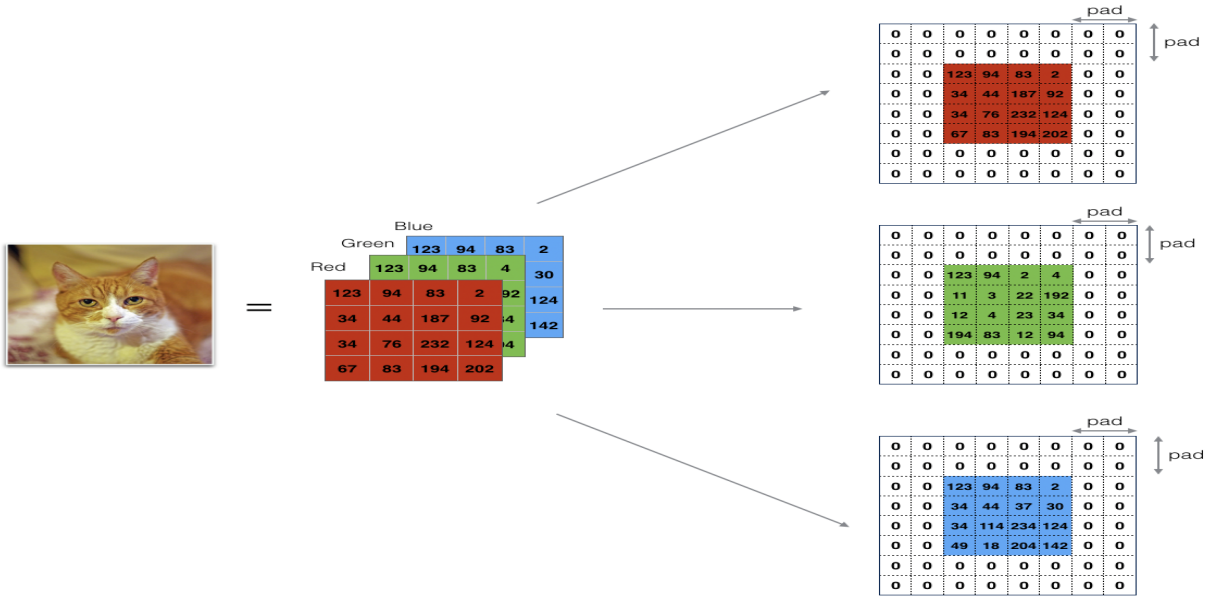


Figure 4 - RGB Input with Padding of 2 [14]

2.1.3. ReLU Layer

The outputs of convolution are then forwarded as inputs to the following block. Which is the Rectified Linear Unit (ReLU). ReLU introduces nonlinearity in order to create complex signals, in order to create complex functions that can correctly classify between features to appropriately infer between them.

In practice it's carried out by applying the Max(0,X) function, shown in figure 5.

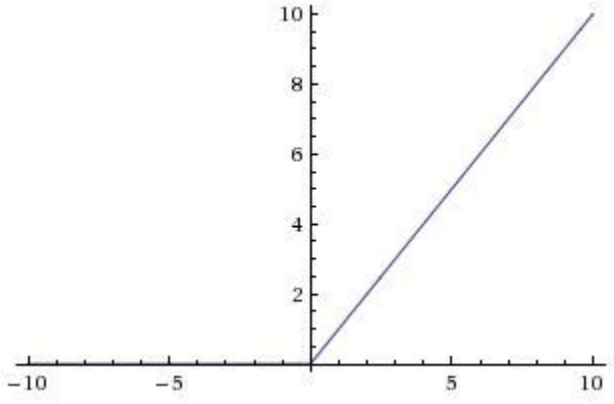


Figure 5 - ReLU function Graph

Moreover, the convolution operation results in increased tensor dimensionality as shown in the below figure.

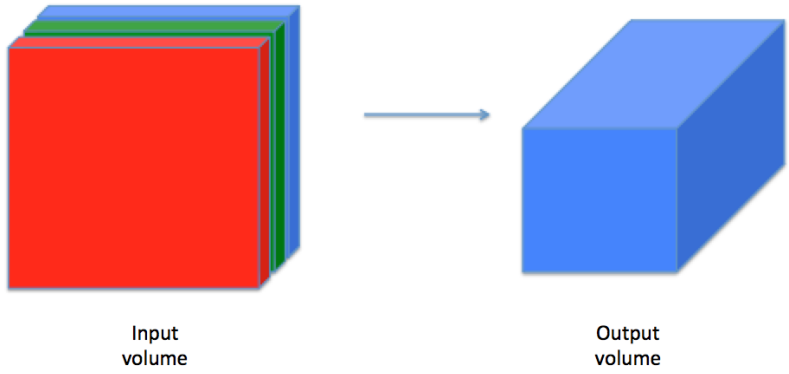


Figure 6 - Convolution Output Depth [14]

2.1.4. Pooling Layer

This added weight can be overcome by the pooling layer; pooling or nonlinear downsampling can reduce computational complexity as well as memory footprint by downsampling the height and width of the input.

Pooling slides a window of shape f x f with a specified stride value as can be seen in Figure 5. This allows the kernels to store the same feature maps but in downsampled form.

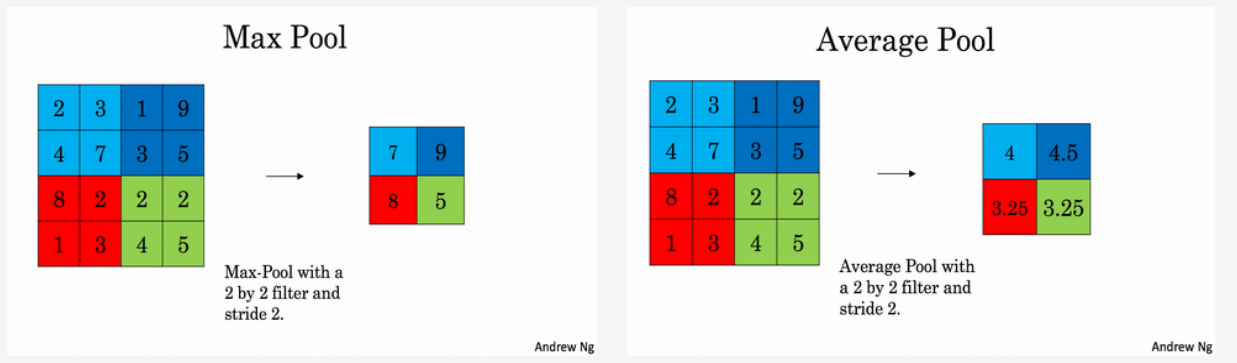


Figure 7 - Pooling Types [14]

After applying the pooling operation, the output requires a preallocated memory of size (#images) x (n_H) x (n_W) x (image depth) where n_H and n_W are preprocessed values obtained by using Equation 2.

$$n_H = \lfloor \frac{n_{H_{prev}} - f}{stride} \rfloor + 1$$

$$n_W = \lfloor \frac{n_{W_{prev}} - f}{stride} \rfloor + 1$$

$$n_C = n_{C_{prev}}$$

Figure 8 - Processing of Pooling Output Dimensions [14]

2.2. Platforms

Edge devices are now gaining more computing power, which provides the possibility to use more complex multi-layer convolutional neural networks to implement powerful deep learning applications. It is difficult to use CPU platforms for different deep neural network applications because they cannot provide sufficient parallel computing power. GPU platforms are considered the first options used for deep neural network operations due to their highly parallel design.

Besides CPUs and GPUs, FPGA-based accelerators have become a researching topic for deep neural network applications which is a possible solution to exceed GPUs, because they are showing high-speed operations and achieving energy-efficient neural network processing compared to CPUs and GPUs. FPGAs introduce high parallelism so they support excluding additional logic by using the properties of neural network computation without hurting the accuracy. Deep neural network frameworks like TensorFlow and Caffe are not existing for FPGAs platforms, Which implement deep neural network models that are much harder than that on CPUs and GPUs [13].

2.2.1 GPU/CPU

GPU platforms are considered as a unitary processor, GPUs are hardware accelerators in vector/tensor computational potential, designed at evaluating three/four dimensional vectors to provide 3D graphics rendering in real time. In [10] the author uses the MNIST handwritten data sets for testing a convolution neural network (CNN) with 200 hidden layers. After testing the performance of the GPU was found to be better than CPUs and more efficient. Their results show that their GPU platform (NVIDIA GeForce 6800 Ultra) leveraged a threefold increase in performance over the CPU platform (Intel 3GHz P4). These results show the superiority of the CNNs processing on GPU platforms over CPU platforms.

2.2.2. FPGA/GPU

FPGAs are the hardware that can dynamically be configured by hardware engineers using hardware description language (HDL). In [11] the author shows that the FPGA Platforms afford better performance than GPU platforms per watt, where the results show that FPGA often speeds up the performance by 3-4X compared to GPU. The Arria 10 GX1150 FPGA systems were capable of processing 233 images per second at a device power of 25 watts. NVIDIA K40 GPUs, on the other hand, could process 500-824 images per second at a device power of 235 watts [3]. Another experiment was done using different platforms to compare the differences in performance between FPGA and GPU comparing the performance of Nvidia's (Jetson TX2 CPU-GPU), a renowned embedded systems board with Intel's (Altera Cyclone10GX FPGA). With platforms running different CNN models SqueezeNet, MobileNetv2, and ShuffleNetv2. The results show that FPGA surpasses GPU over MobileNetv2 with our most important metrics being power efficiency and latency FPGAs showed 30% power decrement and 4% to 26% subtraction in latency, whereas in ShuffleNetv2 they uncovered a power decrement of 25% and a latency decrement of 21%, and SqueezeNet (21%-28% energy subtraction , same latency) [12].

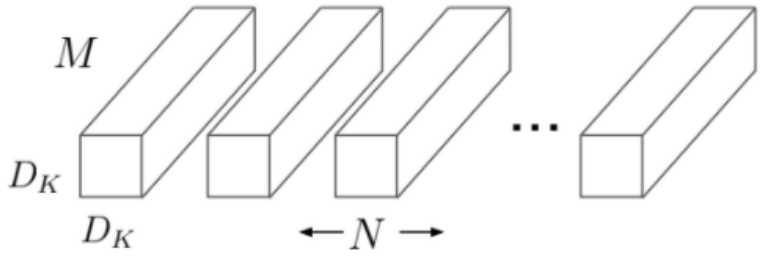
2.3. Model Architectures

In order to approach the implementation in a lightweight manner, every part of the project aims to achieve a balance between efficiency and accuracy, thus Mobilenet models are researched in order to outline their advantages and drawbacks as compared to standard convolutional models. MobileNet model uses Depthwise Separable convolutions, these types of separable convolutions epitomize the standard convolution used by models such as Alexnet, but in a reduced form. This is achieved by splitting the standard convolutional operation into 2 parts; the depthwise convolution and the pointwise convolution.

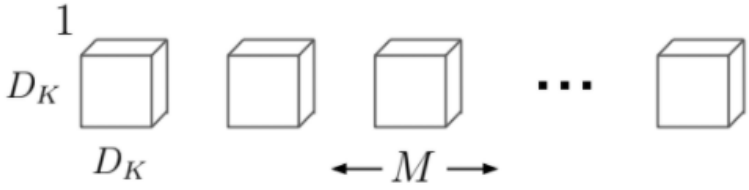
The standard convolution, which is applied in Alexnet models; works by filtering features using convolutional kernels and then combining them to create new representations. In a more concrete sense, the standard convolution applies an element wise multiplication followed by a depth-wise summation. By using depthwise separable convolutions, designers can benefit from even more increased parallelization of the convolution operation by pipelining the convolution operation via

the use of depth wise convolution to output the feature maps by imposing each kernel to every input channel and the pointwise operation which applies a 1x1 convolution to combine the maps. Achieving small parameter sized, low latency models which will greatly aid in achieving this project's application which abides to tight requirements.

Not only does Mobilenet introduce optimized convolutions, but they also introduce a set of two hyper-parameters which further reduce model size and computational complexity.



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters

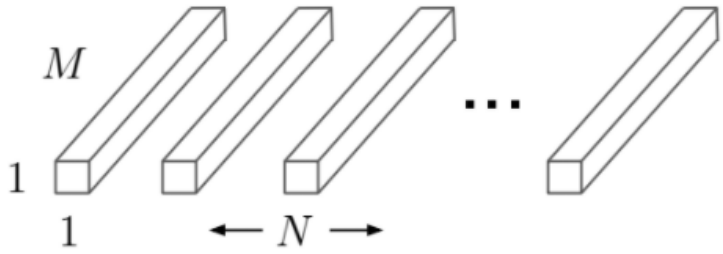


Figure 9 - Output Depth of Standard vs Depthwise Convolutions [15]

Complexity of Standard convolution operation:

$$D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F$$

Complexity of Depthwise convolution:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F$$

Complexity of Depthwise Convolution + Pointwise convolution:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F$$

Depthwise separable convolution over standard convolution complexity reduction:

$$\frac{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F} = \frac{1}{N} + \frac{1}{D_K^2}$$

Thus, as shown, depthwise separable convolution is objectively mathematically less complex than standard convolution.

The depthwise convolution used by MobileNets imposes singular kernels to each image depth, followed by a batch normalization and a ReLU block. The depthwise convolution's outputs are then combined using a 1x1 convolution by the pointwise convolution.

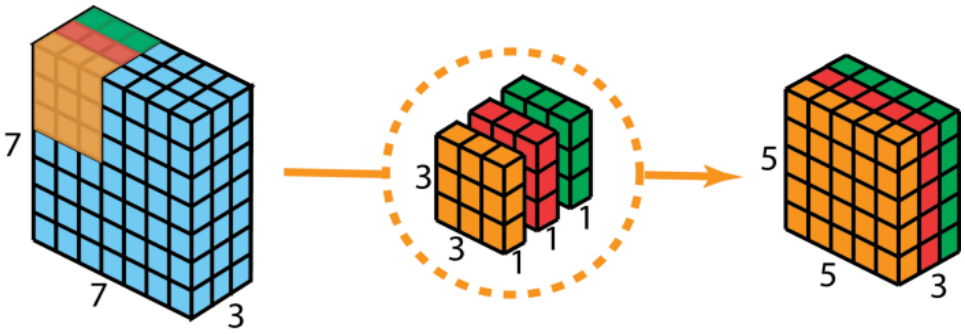


Figure 10 - Depthwise Output [16]

Pointwise convolution reduces input depth, compared to standard convolution as shown in figure 11, this operation uses 8 to 9 times lower computation over Alexnet like implementations, while sacrificing little loss in accuracy.

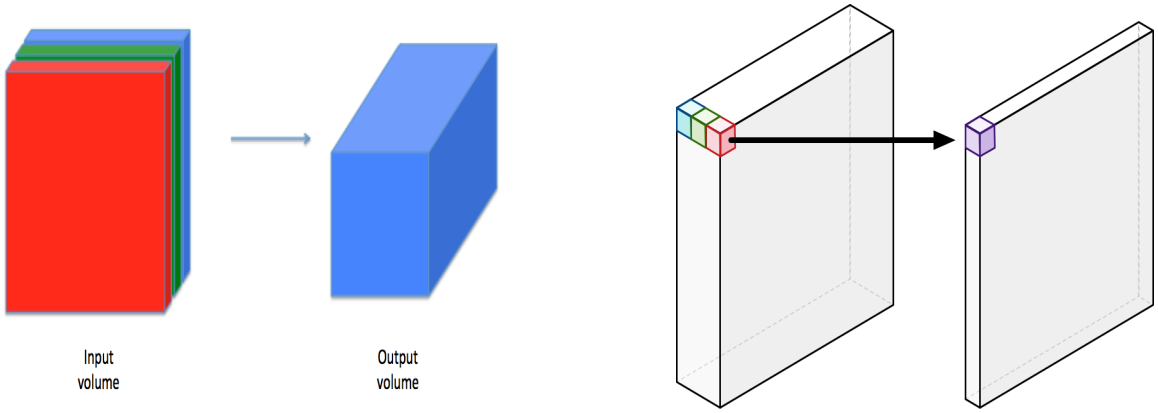


Figure 11 - Pointwise Convolution Condensed Output [14] [16]

In order to accommodate platforms like FPGAs that leverage faster computation and lower power efficiency over limited resources, engineers have access to MobileNet's hyperparameters which aim to reduce model input and parameter sizes. Shown below is the depthwise separable computational complexity, appended on to it is α (width multiplayer) and ρ (resolution multiplier). These variables can fundamentally reduce computational cost and resolution of parameters. Both are used in our own Mobilenet implementations.

$$D_K \cdot D_K \cdot \alpha M \cdot \rho D_F \cdot \rho D_F + \alpha M \cdot \alpha N \cdot \rho D_F \cdot \rho D_F$$

2.4. CNN on FPGA

After reviewing lightweight models that achieve desirable attributes; focus was shifted towards reviewing the tools that allow FPGAs to run these models for quick, efficient and on the edge implementations.

2.4.1. HLS

Out of 33 papers that were collected for an implementation review, it was noted that approximately 75% of the reviewed implementations opted for using high level synthesis (HLS) tools in order to quickly iterate upon their implementations and avoid low level designs. However, this approach leverages scalability and short design times with utilization inefficiency, less design control and lower throughput compared to RTL approaches. An HLS approach is reviewed as a basis to allow understanding into FPGA interfacing and CNN FPGA designs, which will provide a stepping stone into more low level implementations.

2.4.1.1 Vivado

HLS allows for a high level approach for design by using C-like languages which can fully synthesize the accelerated function into multipliers, counters, ect like in the case of Vivado for Xillinx and Quartus for Altera.

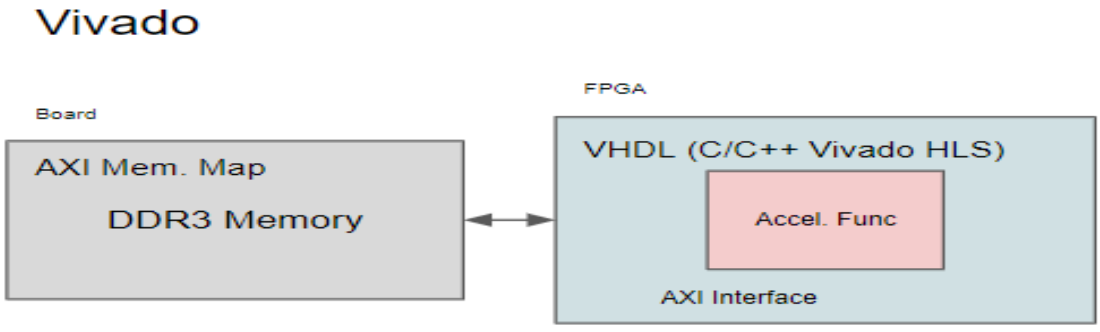


Figure 13 - Vivado Block Diagram

2.4.1.2. OpenCL

Choosing to follow a more guided approach where specific code cells are indicated by the designer using preprocessor pragmas (compute kernels), so that the HLS compiler can assign specific FPGA blocks to it, like with OpenCL.

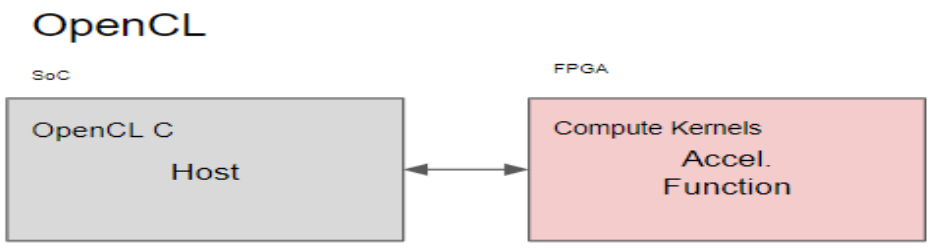


Figure 14 - OpenCL Block Diagram

2.4.1.3. Raw Implementation

The third approach is called the Raw implementation throughout this report. This approach allows Python to directly run on the hard processor system (HPS), when flags are called, the FPGA (programmable logic) blocks carry out computations via RTL synthesis and return them to the HPS via AXI/Avalon memory mapped interfaces. This approach can be accomplished via the use of Intel Altera devices that support HPS components.

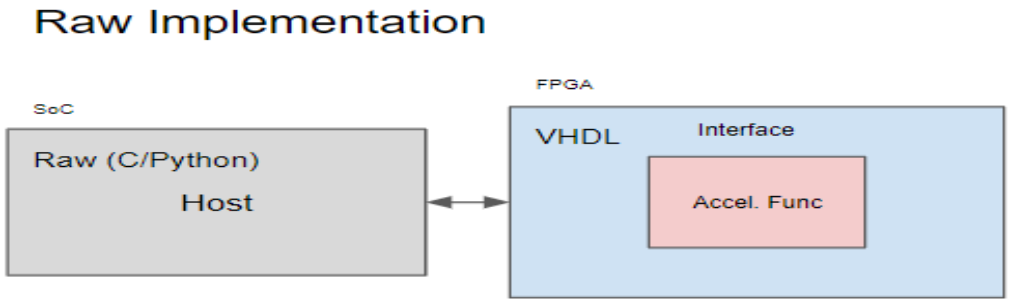


Figure 15 - Raw Block Diagram

Introducing comparison tables between each implementation will allow for contrast on the leverage between complexity and efficiency.

2.4.1.4. General Attributes for HLS Tools

Following is a general attribute table for the researched CNN implementations on FPGA. This allows us to focus our efforts on specific tools rather than a more obtuse approach.

Table 4 - HLS General Table

Metric	OpenCL	Raw SoC	Vivado	RTL
Host lang	OpenCL C	C/Python	C/C++	VHDL
Accel. lang	Compute Kernel	VHDL	C/C++	VHDL
Host interface	SDK/DMA	Avalon/AXI/ DMA	Avalon/AXI/ DMA	DMA/Not required depending on model size

2.4.1.5. Utilization Table

Following is a utilization table, where different implementations are aggregated.

Table 5 - HLS Utilization Table

Metric	OpenCL [6]	Vivado [7]	RTL [8]
Device	DE5-Net	xczu9eg-ffvb1156-2-i	Virtex VC709
ALUT/ALMs	49%	11%	66%
FF	-	6%	31%
BRAM	74%	5%	68%
DSP	100%	5%	57%

2.4.1.6. Performance Table

Following is a performance table

Table 6 - HLS Performance Table

Metric	OpenCL [6]	Vivado [7]	RTL [8]
Device	DE5-Net	xczu9eg-ffvb1156-2-i	Virtex VC709
Precision	8 bit	11 bit	16 bit
# Op. per Image	1.46 GOP	410,758 Cycles	611.52 GOP
Inference Time (Approx.)	45.7 ms	3.58 ms	2.41 ms

It is important to note that Vivado [7] implementation successfully implements the input layer and one convolutional layer. By extrapolation; it can be deduced that the RTL [8] will yield more efficiency and extract higher throughput. An important exploration into CNN RTL implementation will provide needed insight into executing a bleeding edge implementation.

3. MNIST VHDL Implementation

Nowadays there are remarkable achievements of the convolutional neural networks in the fields of recognition and classification, but such tasks require a huge number of intensive computations. For that reason increasing the speed of the neural networks is an important target to meet. In order to speed up the heavy computations of CNNs, engineers and researchers proposed FPGA hardware accelerators due to good performance, high energy efficiency. The CNNs can be developed on FPGAs using either hardware description language (HDL) or high level synthesis (HLS). Using the HDL based architectures is preferred due to its high performance concerning good performance, high energy efficiency, frequency, latency, and resources utilization. The proposed CNN architecture was developed on an FPGA using VHDL for the application of the MNIST dataset. The CNN architecture is composed mainly of four layers, the input layer, the convolution layer, the pooling layer and the fully connected layer (Kyriakos, 2019).

3.1. Input Layer

As an initial step, the input layer in the implemented convolutional neural network, is mainly used to receive the values of an input image with 6x6 pixels' values as well as receiving the values of the 3x3 kernel both from external files. Each of the values of the image and the values of the kernel will be then stored in separate ROMs. Meaning that the two ROMs will be initialized from the external files by using HDL. Each line in the two initialization files represents a value stored in the ROMs. The contents of the ROMs can be represented either in hexadecimal or binary meaning that the number of lines in one initialization file should be equal to the number of rows in the corresponding ROM. Also it is important to mention that the contents of the external file should be written as pure binary or hexadecimal, no additional information or comments are allowed to be written. The extension of the used initialization files are for example (filename.data) [20]. The implementation of the input layer also includes a counter that is considered the controlling unit of the implemented CNN.

This architecture is implemented to receive gray scale input images. Such images contain only one color channel, so only one 3x3 kernel is needed in the design. The main task of the input layer is to create all the convolutional windows needed to perform the convolution operation with the 3x3 input kernel.

20	10	15	25	10	0
2	32	41	5	32	1
54	26	19	2	26	22
6	72	11	86	72	18
16	23	5	0	23	1
6	72	11	86	72	18

Figure 16 - The values of the pixels' intensities of the given input image

-1	0	1
0	-1	0
0	-1	1

Figure 17 - The values of the filter kernel

3.2. Convolutional Layer

The second layer in the implemented CNN architecture is the convolution layer. This layer mainly consists of DSP blocks. Each DSP block includes multipliers, tree of adders, and a ReLU block. DSP blocks are used to perform the convolution operation between the pixels' values of the input image with an input kernel of size 3x3. The convolution operation is calculated by mathematically performing dot product operations on every 3x3 convolution window with the input kernel. Only one result is expected as an output of each convolution operation. The number of convolution windows is equal to $(Image\ Size\ -\ Kernel\ Size\ +\ 1)^2$. After completing the convolution operation for all the convolution windows, a bias value is added to each of the output results, then these results will be forwarded to the ReLU activation function.

The output values of the convolution layer are stored in a memory RAM then forwarded to a 16x4 encoder that receives the sixteen output values and forward them as four sets of data to the pooling layer. The ReLU block executes the following function:

$$ReLU(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

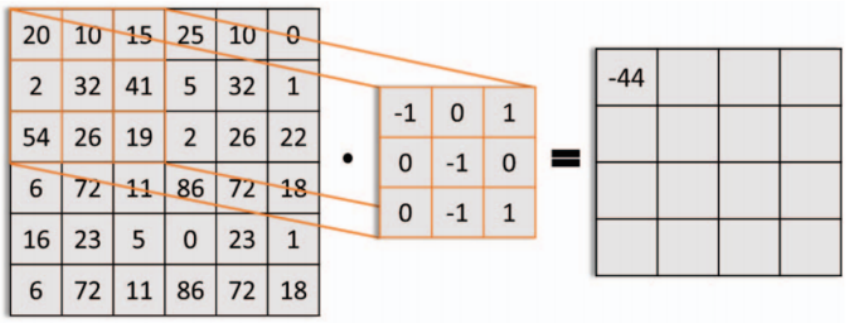


Figure 18 - The convolution operation of image's pixels

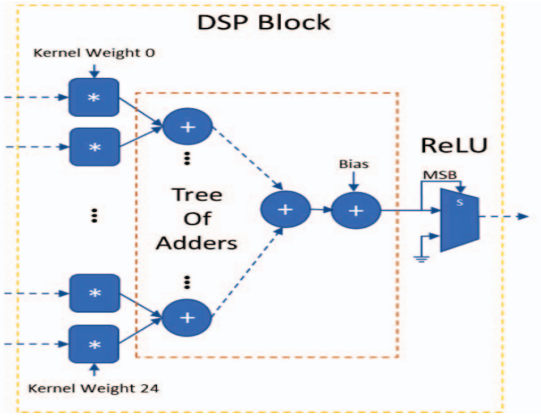


Figure 19 - The architecture of the convolution of DSP block

-44	-43	14	-61
-48	29	-25	-84
-125	-40	-56	-74
-79	84	47	-145

Figure 20 - The output matrix of the convolution operation

0	0	15	0
0	30	0	0
0	0	0	0
0	85	48	0

Figure 21 - The output of the convolution layer after adding the bias value and passing the values to the ReLU activation function

3.2.1. Functional Simulation

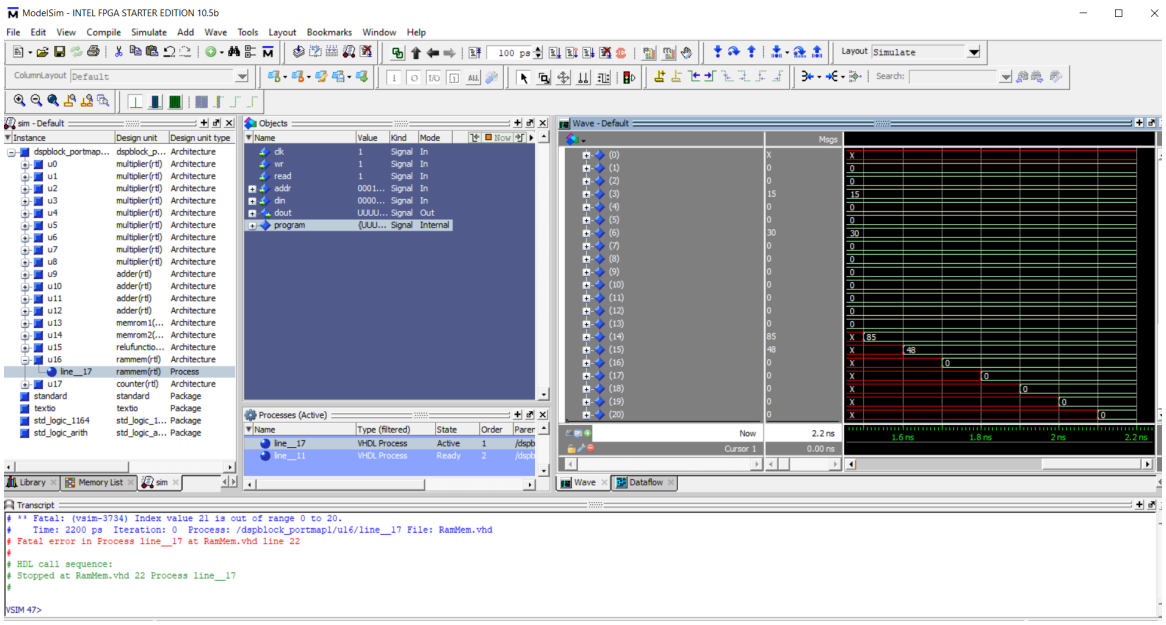


Figure 22 - Functional simulation of the convolution layer

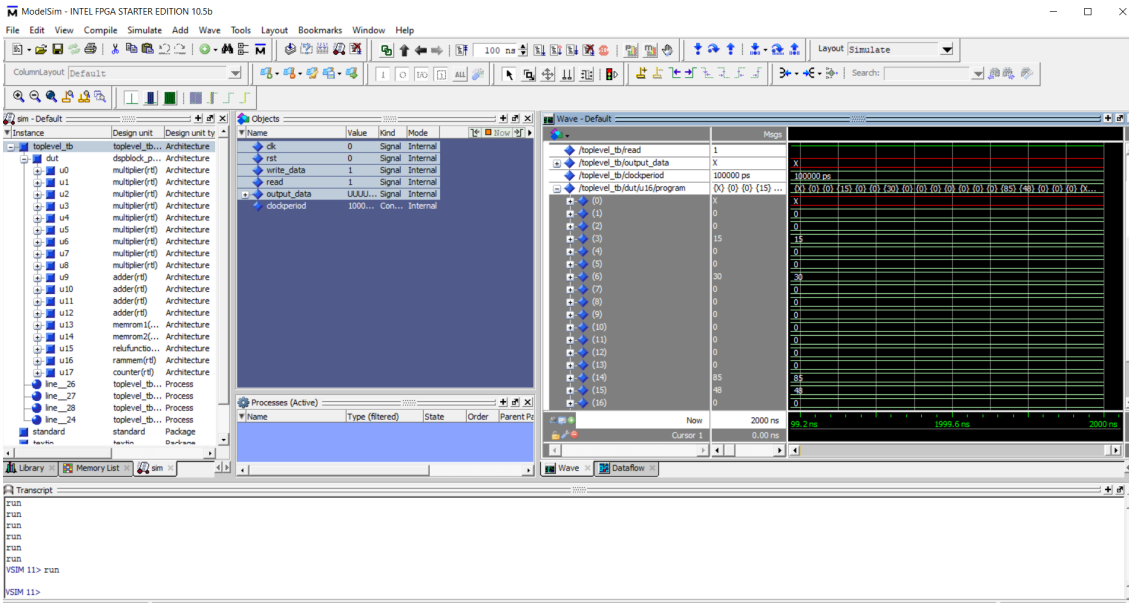


Figure 23 - Functional simulation of the test-bench of the convolution layer

3.3. Pooling Layer

The third layer in the architecture is the pooling layer, it works as the bridge between the convolution layer and the fully connected layer. It mainly consists of pooling blocks. Each pooling block includes a Demux finite state machine that forwards the values from the corresponding DSP block as four sets of values to the pooling FIFOs through 4x16 decoder. The Pooling block also includes pooling FIFOs and Max-pooling finite state machines. The pooling FIFOs are mainly used to store and forward four sets of values from the Demux FSM to registers. Each register will hold the intensity value of one pixel. Finally, the main task of the max-pooling finite state machines is to extract the greatest value of four pixels of 2x2 sized-windows with stride of 2. The following two finite state machines represents the DemuxFSM and the MaxPoolingFSM:

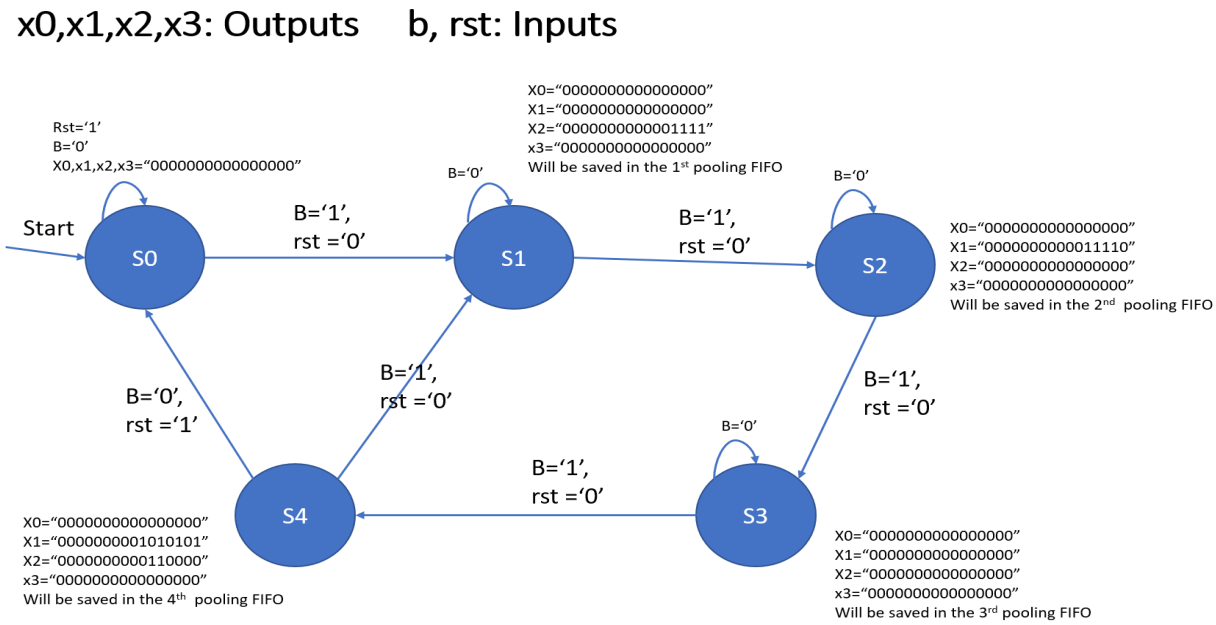


Figure 24 -The finite state machine of the DemuxFSM of the pooling layer

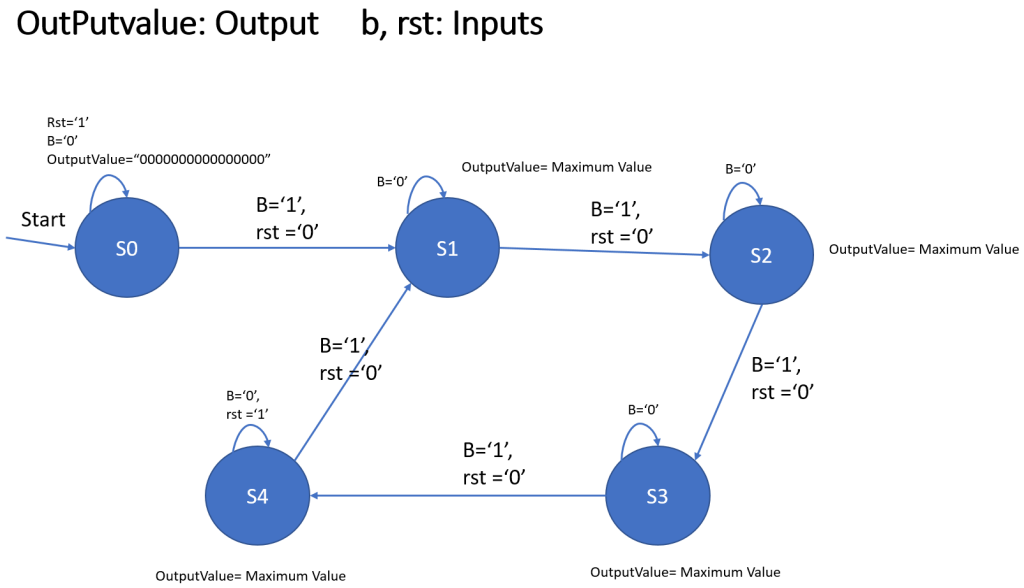


Figure 25 -The finite state machine of the MaxPoolingFSM of the pooling layer

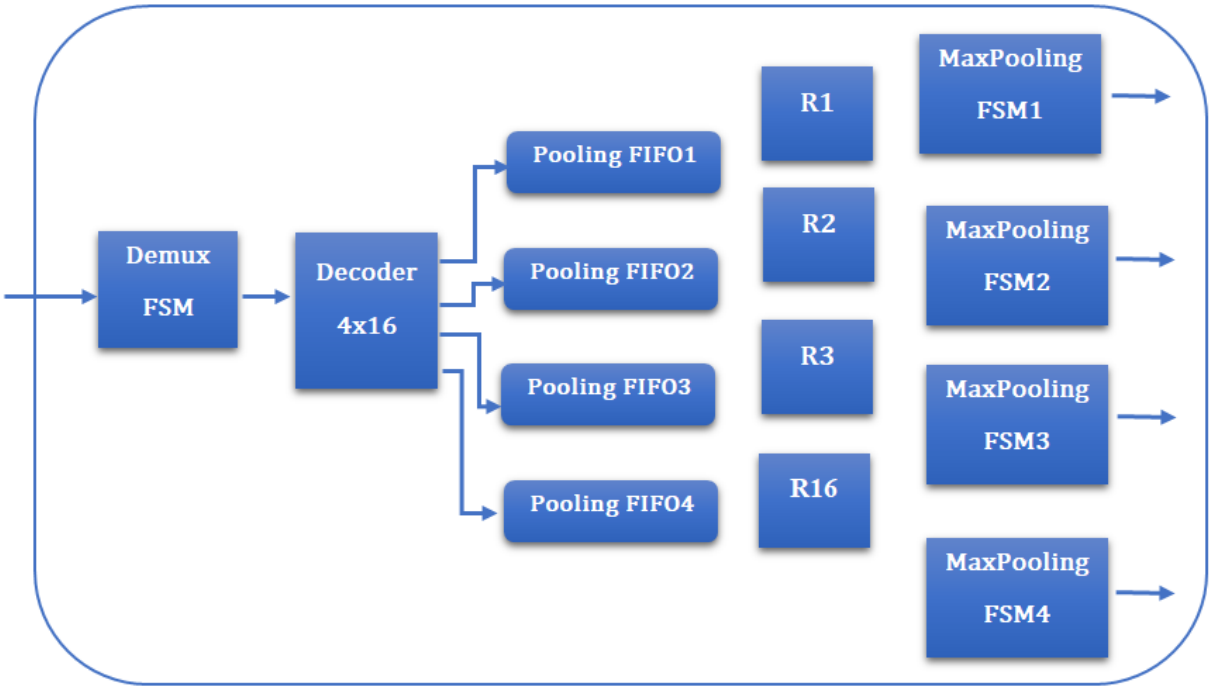


Figure 26 - Pooling Block architecture

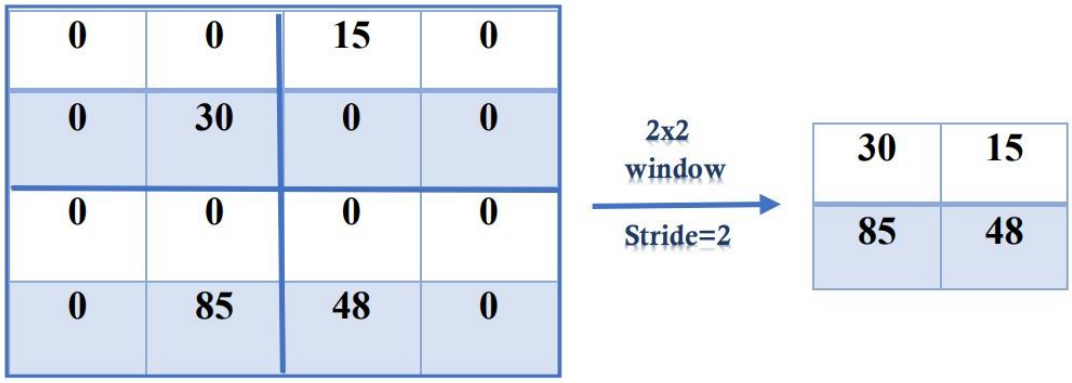


Figure 27 - The output of the implemented pooling layer

3.3.1. Functional Simulation

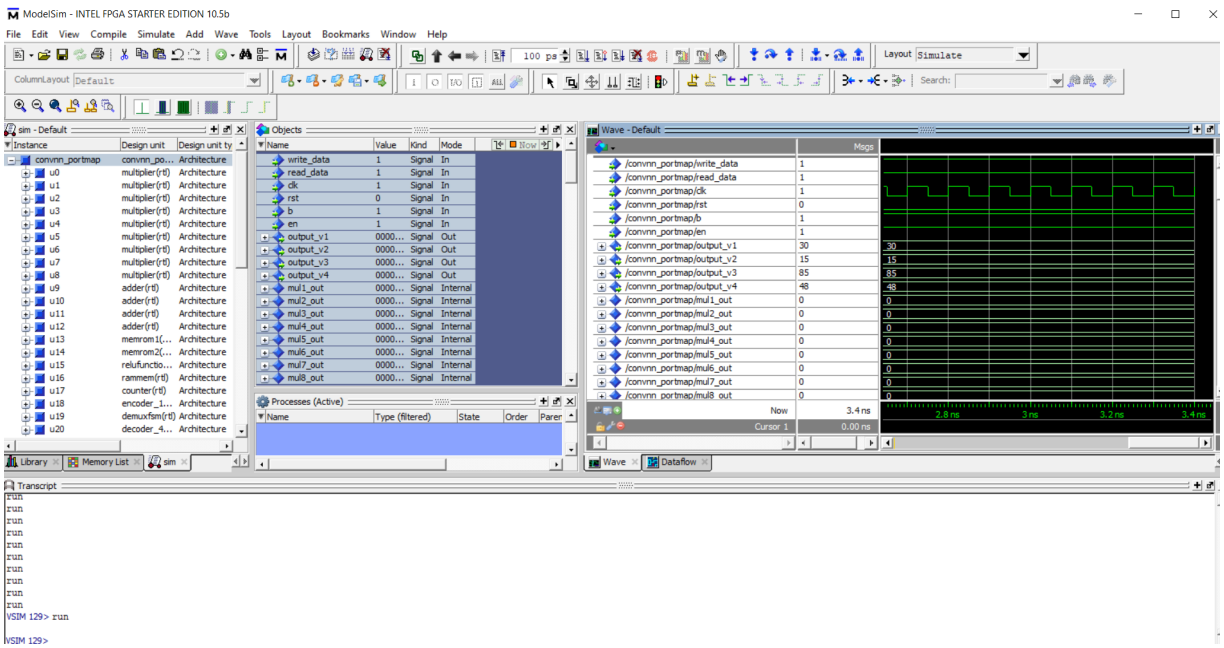


Figure 28 - Functional simulation of the pooling layer

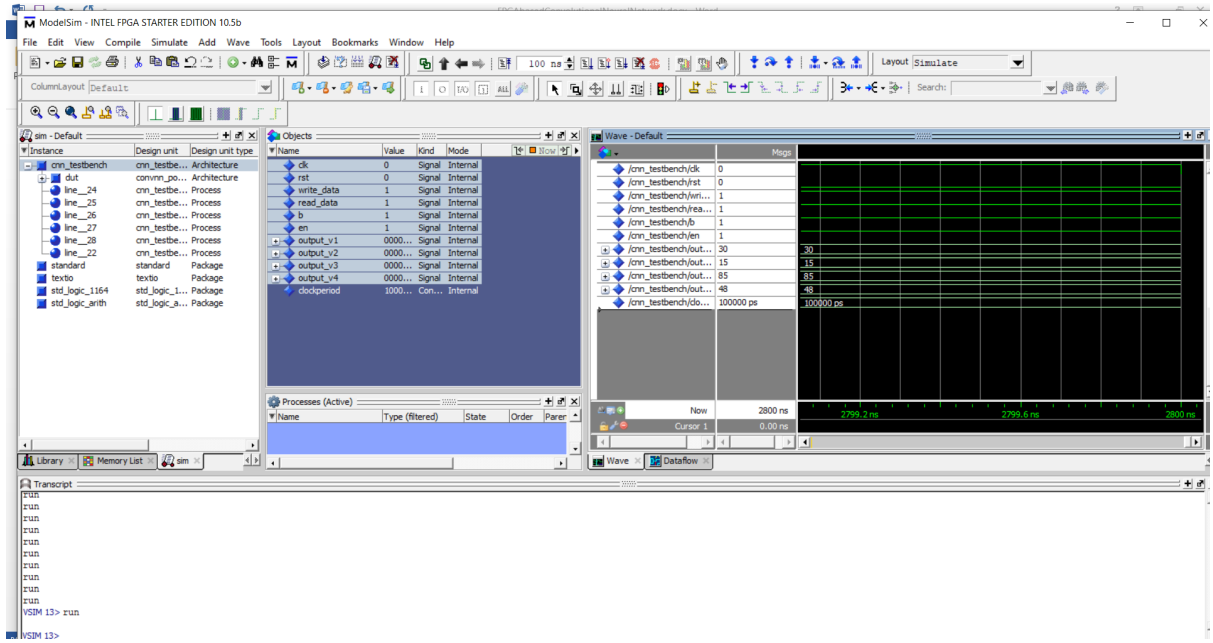


Figure 29 - Functional simulation of the test-bench of the pooling layer

3.3.2. Timing Simulation

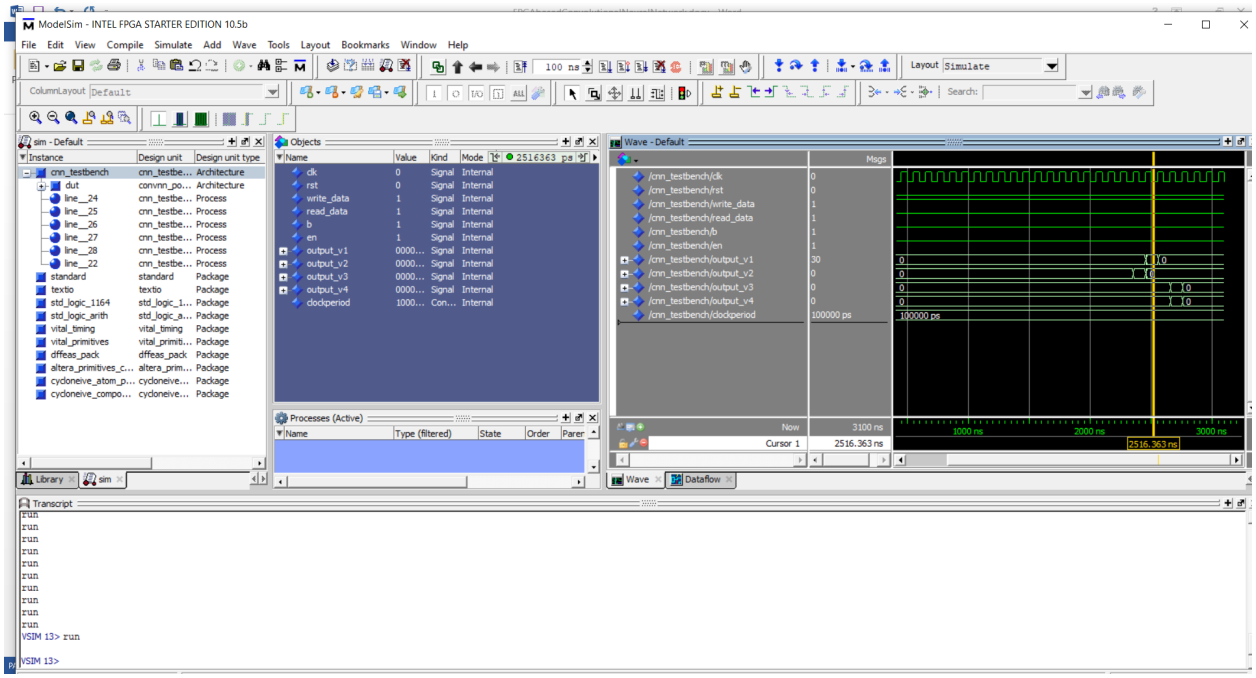


Figure 30 - First output in the timing simulation of the pooling layer

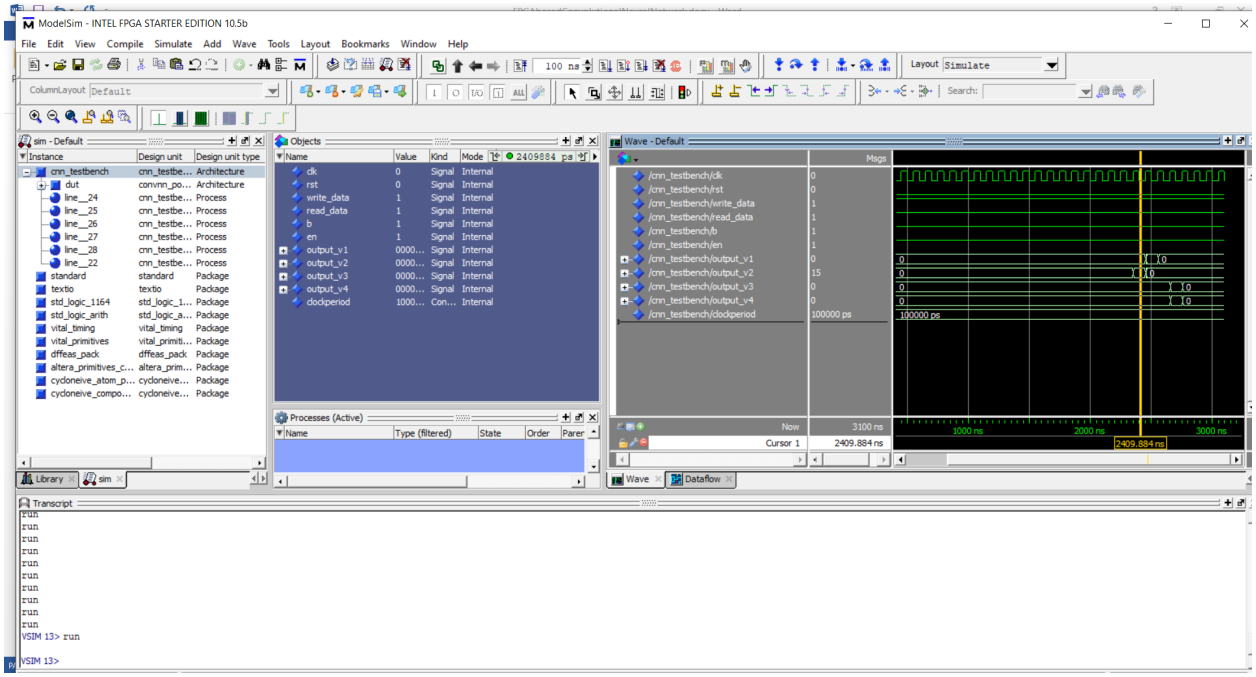


Figure 31 - Second output in the timing simulation of the pooling layer

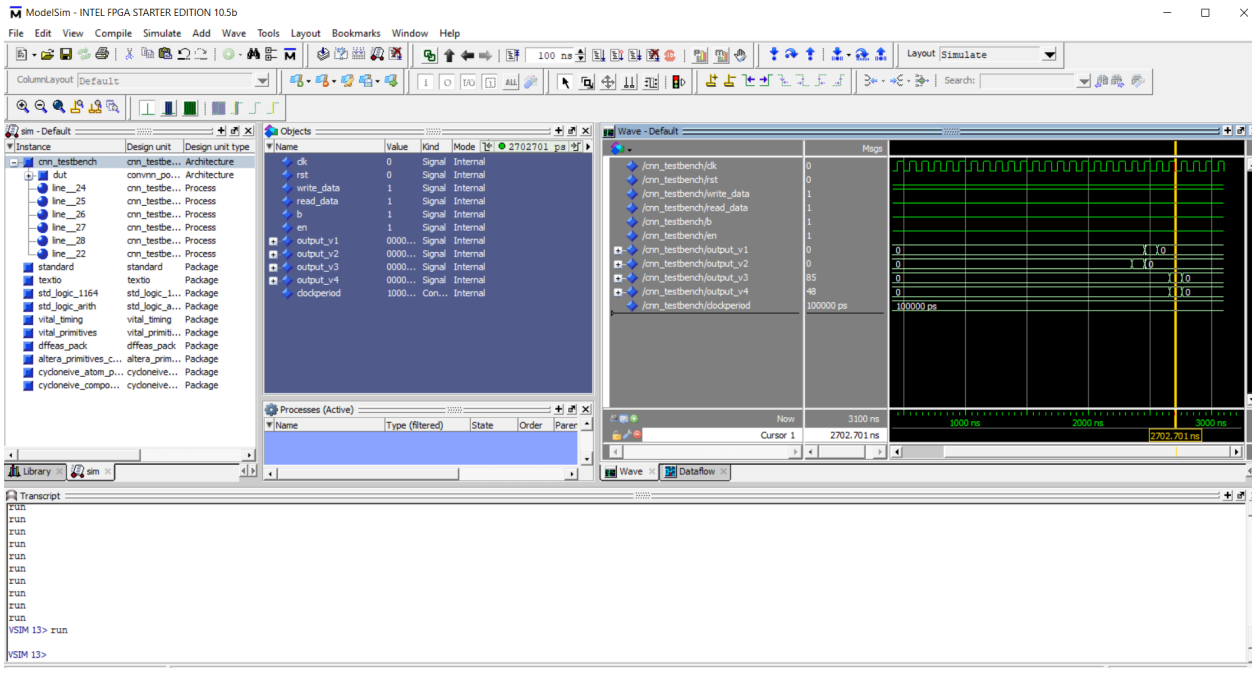


Figure 32 - Third and fourth outputs in the timing simulation of the pooling layer

3.4. Fully Connected Layer

The fully connected layer is the final layer in the implemented CNN architecture. It consists mainly of three blocks, the fully connected layer block, the ReLU block and the output layer block.

1. The fully connected layer block: Fully connected layer block is the vector multiplication component that computes the dot product for two vectors then adding the corresponding bias values to the dot result, the fully connected layer block execute the following equation:

$$O^0 = W^0 . I^0 + B^0$$

Where O^0 *is the output* matrix

W^0 is the weights matrix, assume W^0 = [1 -1 1 -1 0 0 0 0]

I^0 is the output of the pooling layer after flattening, I^0 = [30 15 85 48 0 0 0 0]

B^0 is the Bias vector, assume B^0 = [1 1 1 1 1 1 1 1]

O^0 will be [31 -14 86 -47 1 1 1 1]

1. **The ReLU block:** Each of the fully connected layer block output results is forwarded to a ReLU block

$$I^1 = ReLU(O^0)$$

I^1 = [31 0 86 0 1 1 1 1]

2. **The output layer block:** At first the output layer block computes the multiplication of the block's input vector I^1 which holds the outputs of the previous ReLU block with the weight matrix W^1, then adding the bias vector B^1 to the product of the multiplication and the result is the k^{th} output neuron where k=0,…..,9.

This procedure is executed 10 times, once for every output neuron:

$$O_k^1 = \sum_{k=0}^{9} W_k^1 . I^1 + B_k^1$$

Where O_k^1 *is the output* of the softmax function

W^1 is the weights matrix, assume $at\ k = 0, W_k^1 = [1 \quad -1 \quad 1 \quad -1 \quad 0 \quad 0 \quad 0 \quad 0]$

I^1 is the output of the ReLU activation function $I^1 = [31 \quad 0 \quad 86 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1]$

B^1 is the bias vector, assume $B^1 \ at\ k = 0$ is equal to 1

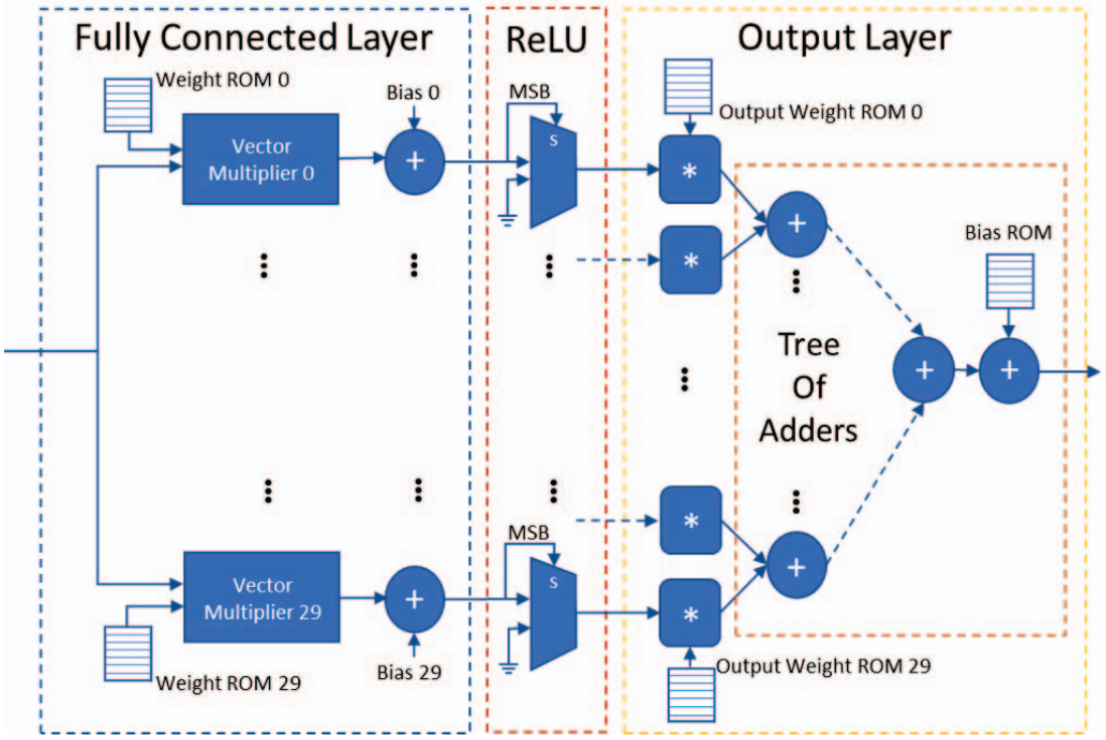


Figure 33 - Fully connected layer architecture

3.5. The Implemented CNN Model Block Diagram

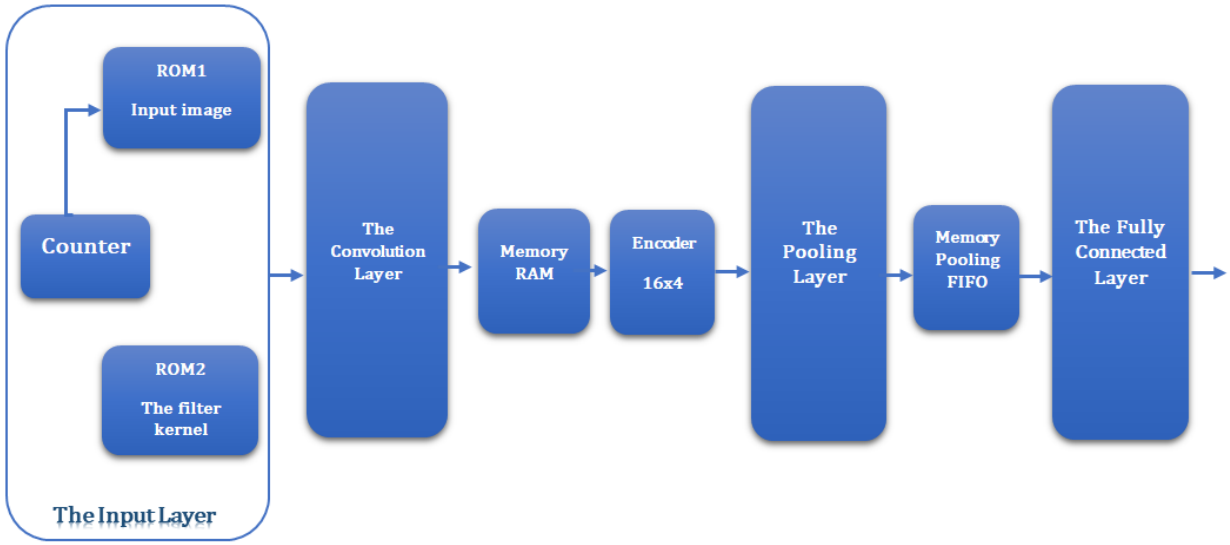


Figure 34 - The block diagram of the implemented CNN architecture

3.6. Overall Design Functional Simulation

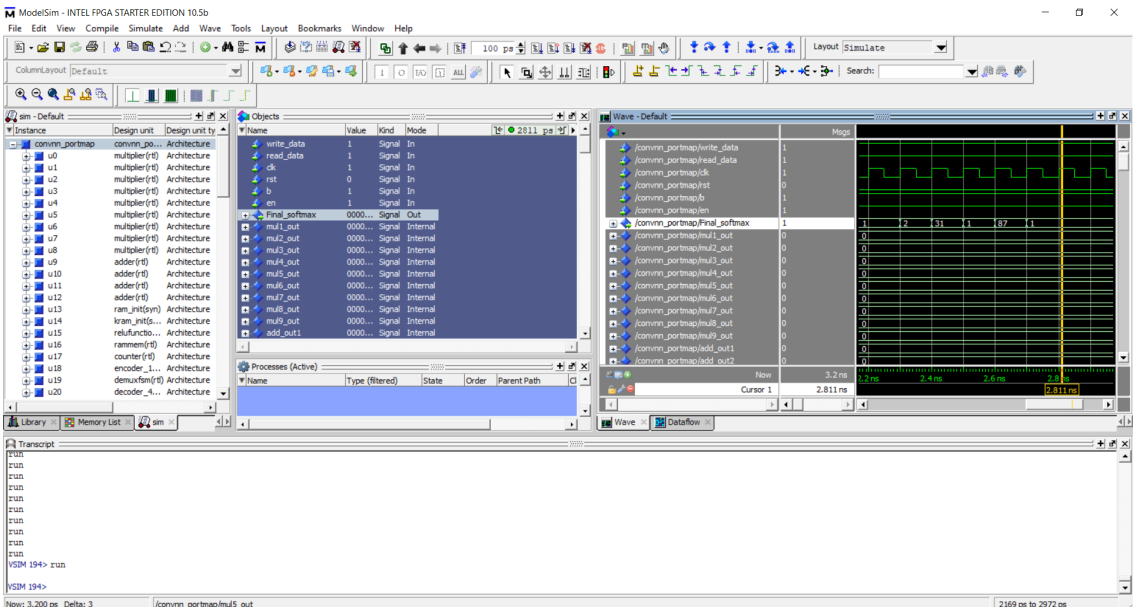


Figure 35 - Functional simulation of the overall CNN design

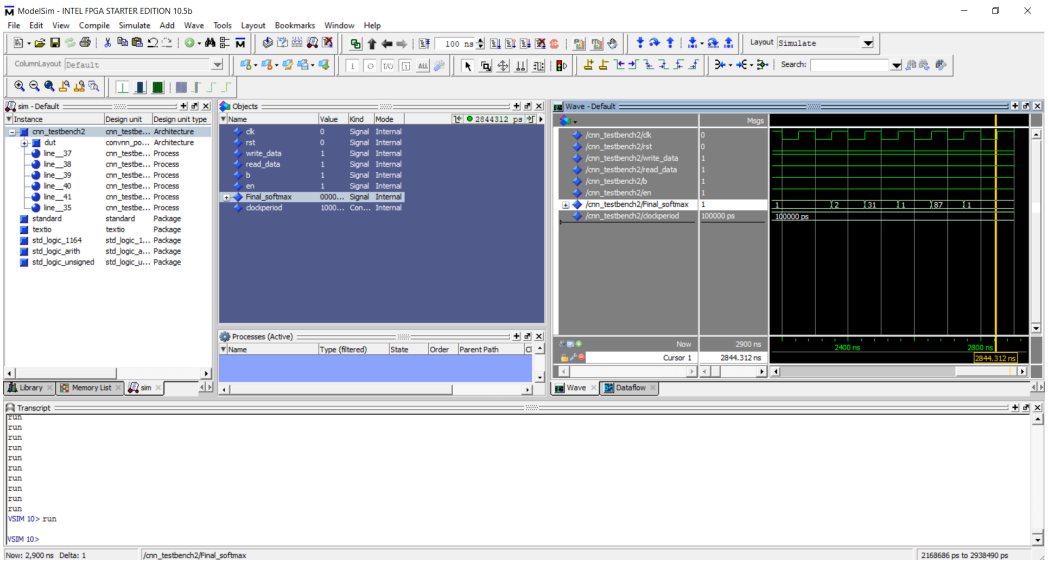


Figure 36 - Functional simulation of the overall design test-bench

3.7. Overall Design Timing Simulation

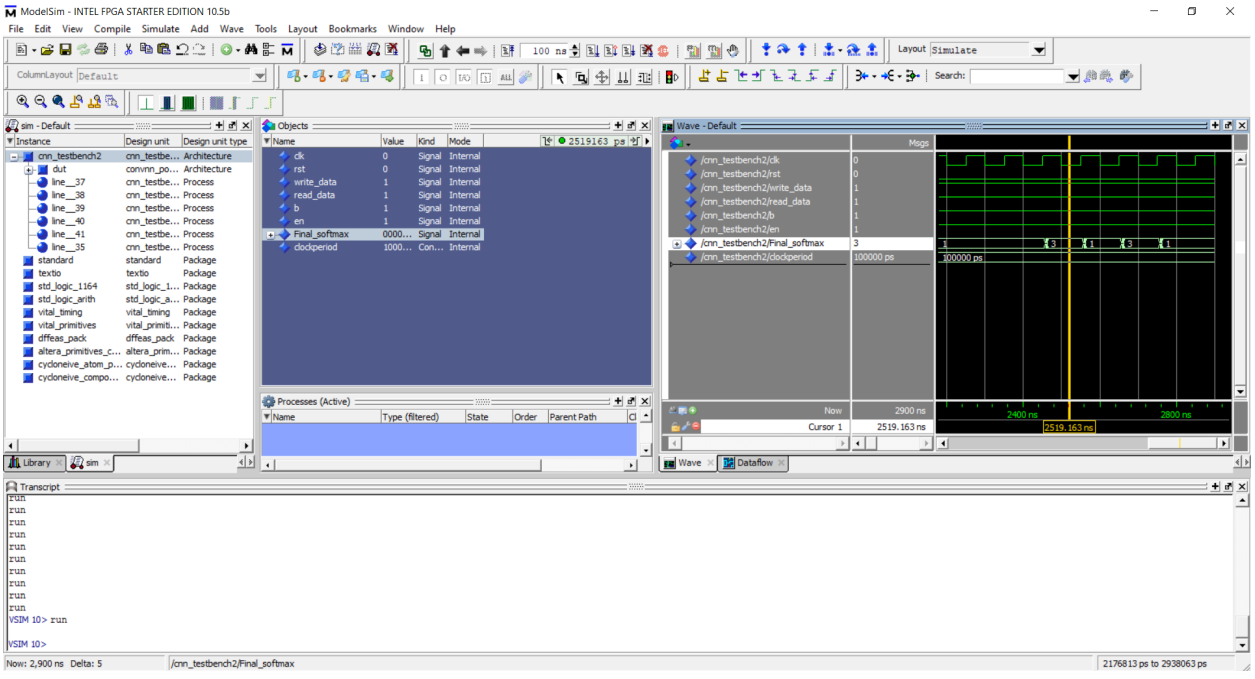


Figure 37 - Timing simulation of the overall CNN design

3.8. The CNN model results, the frequency, clock cycles of the design, and Quartus resources

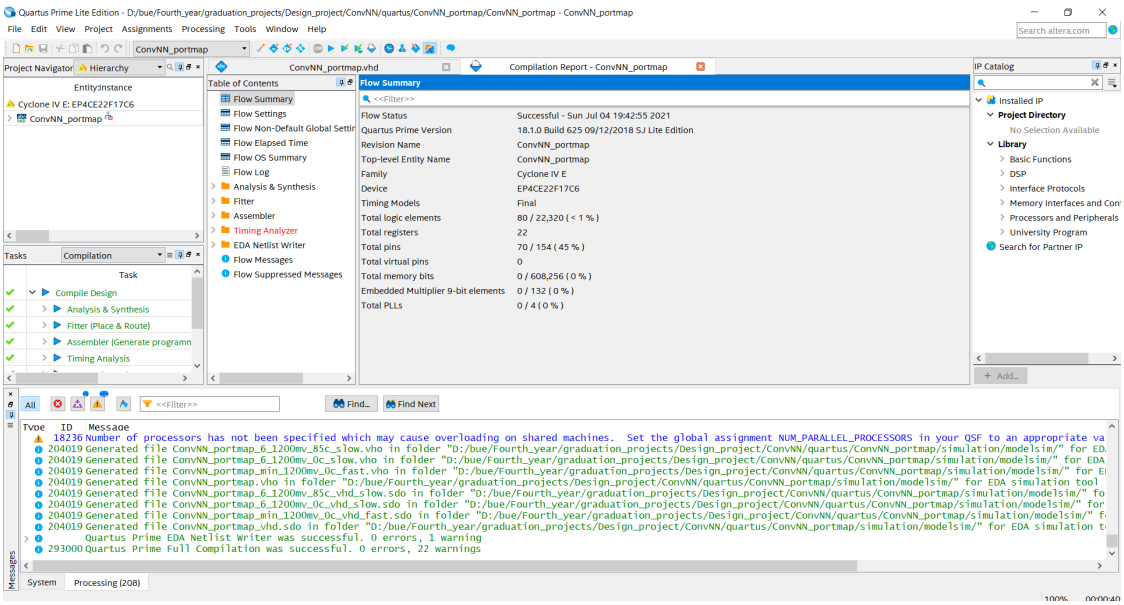


Figure 38 - The Quartus resources of the implemented CNN model

3.9. Design Summary

Table 7 - Flow Summary

Flow Summary	
Flow Status	Successful - Sun Jul 04 19:42:55 2021
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	ConvNN_portmap
Top-level Entity Name	ConvNN_portmap
Family	Cyclone IV E
Device	EP4CE22F17C6
Timing Models	Final
Total logic elements	80 / 22,320 (< 1 %)
Total registers	22
Total pins	70 / 154 (45 %)
Total virtual pins	0
Total memory bits	0 / 608,256 (0 %)
Embedded Multiplier 9-bit elements	0 / 132 (0 %)
Total PLLs	0 / 4 (0 %)

Table 8 - Figure Summary

<table>
<tr><th>Layers</th><th>The output results</th></tr>
<tr><td>The Input Layer</td><td>
<table>
<tr><td>-1</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>-1</td><td>0</td></tr>
<tr><td>0</td><td>-1</td><td>1</td></tr>
</table>
<table>
<tr><td>20</td><td>10</td><td>15</td><td>25</td><td>10</td><td>0</td></tr>
<tr><td>2</td><td>32</td><td>41</td><td>5</td><td>32</td><td>1</td></tr>
<tr><td>54</td><td>26</td><td>19</td><td>2</td><td>26</td><td>22</td></tr>
<tr><td>6</td><td>72</td><td>11</td><td>86</td><td>72</td><td>18</td></tr>
<tr><td>16</td><td>23</td><td>5</td><td>0</td><td>23</td><td>1</td></tr>
<tr><td>6</td><td>72</td><td>11</td><td>86</td><td>72</td><td>18</td></tr>
</table>
</td></tr>
<tr><td>The Convolution Layer</td><td>
<table>
<tr><td>0</td><td>0</td><td>15</td><td>0</td></tr>
<tr><td>0</td><td>30</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>85</td><td>48</td><td>0</td></tr>
</table>
</td></tr>
<tr><td>The Pooling Layer</td><td>Output1= 30
Output2= 15
Output3= 85
Output4= 48</td></tr>
<tr><td>The design’s maximum frequency</td><td>416.67 MHZ</td></tr>
<tr><td>The used Clock</td><td>100ns</td></tr>
</table>

4. Applicational Implementation

In order to clearly document our implementation, we chose to list the items in the order we applied it during the implementation phase. Firstly beginning with our methodology into training a model that meets the application requirements. Secondly, a step back in our implementation was the fact that our trained model used TensorFlow to carry out its forward pass function in order to infer inputs, however, TensorFlow is not supported on the FPGA. Therefore, a custom Convolutional Neural Network implementation was executed in order to allow a forward pass on any device that supports Python, currently the custom CNN implementation supports only the MobileNet V1 architecture. Following the completion of the custom CNN implementation, a System Functional Simulation was carried out, where the whole system and its components were simulated as real time components by using threading libraries in Python, while thread to thread communication used a shared memory approach. The matrix multiplication component was transformed from a C algorithm into HDL by using industry techniques, a custom Single Purpose Processor was designed so that when given inputs and specific commands can execute Matrix Multiplication of the two inputs and return the correct output. Wrapping the custom single purpose processor (SPP) with a custom peripheral that follows our peripheral datasheet for managing the custom SPP was essential for the model located on the Hard Processor System (HPS) to use the accelerated matrix multiplication component on the FPGA. This project utilized memory mapped techniques to interface with the custom peripheral by using the Avalon Memory Mapped Interface. With all components in place, functional and interfacing, a full system implementation was achieved.

The Implementation section will discuss our methodology, the many challenges that were met and the unique approaches generated to overcome them.

4.1. Model Training

In order to carry out our CNN fundamental and architectural understanding, we must look into training a CNN model. To allow for a modular approach, a custom model training, validation and evaluation suite was implemented on Google cloud servers to allow for parallel, quick and reliable model creation. 36 models were trained with the use of TensorFlow/Keras libraries to reach an accuracy of 98~99% inside the Google cloud suite. However, this report aims to place focus on six models for discussion and analysis.

The model training suite was executed via the use of Google Colab, a collaborative Python coding environment that allows the use of Google cloud CPUs, this provides increased performance, however due to Google's cost/benefit analysis they chose a 12 hour session timeout. This proved challenging due to the nature of Deep Neural Network training, models take multiple epochs which translates to multiple days to reach an acceptable level. A custom workaround was designed whereby the suite user can choose the number of epochs that the model would train as a variable and after completion the model would save its state, whereby if more training is required, the user would load the latest model state and continue on with the training on a new 12 hour session. Having the intermediate states be saved on Google Drive provided a fully cloud based environment. Not only is a latest snapshot of the model saved, but also every epoch's weights are saved as well.

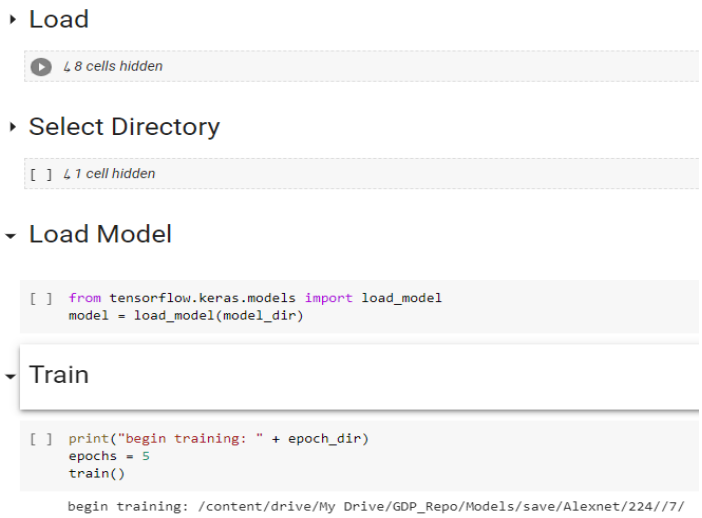


Figure 38 - Model Training Suite

In order to verify an accepted state, for each epoch we recorded the accuracy achieved and the loss incurred. Shown in Figure 39 & 40, are the accuracy and loss graphs for our smallest storage and processing footprint model of MobileNet V1 with 128 resolution and 0.25 width multiplier.

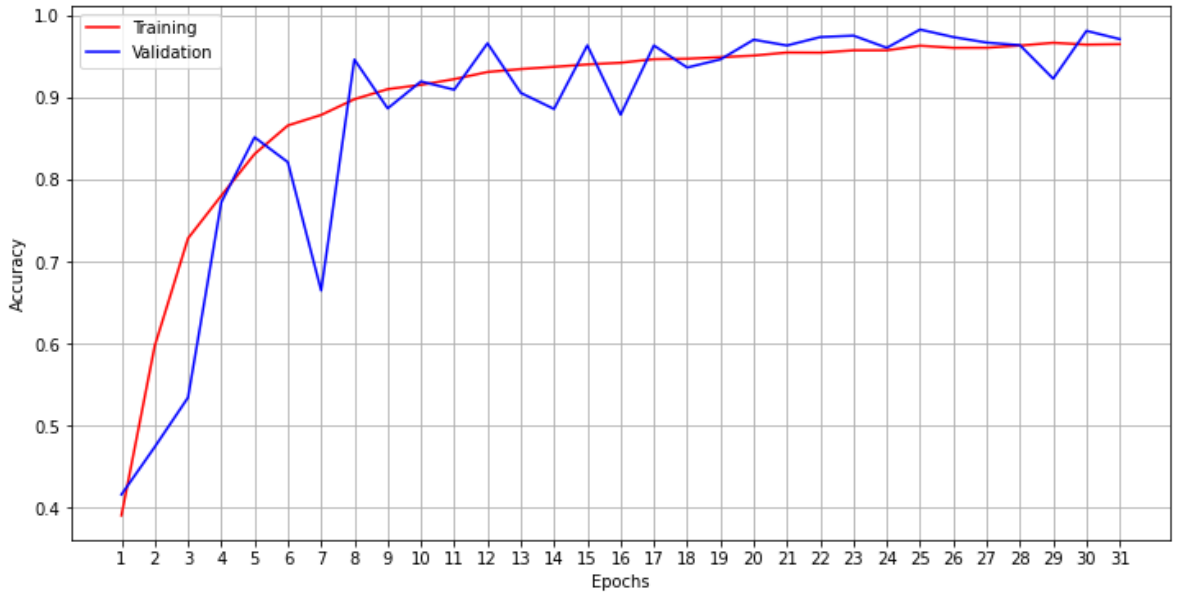


Figure 39 - Epoch/Accuracy Graph

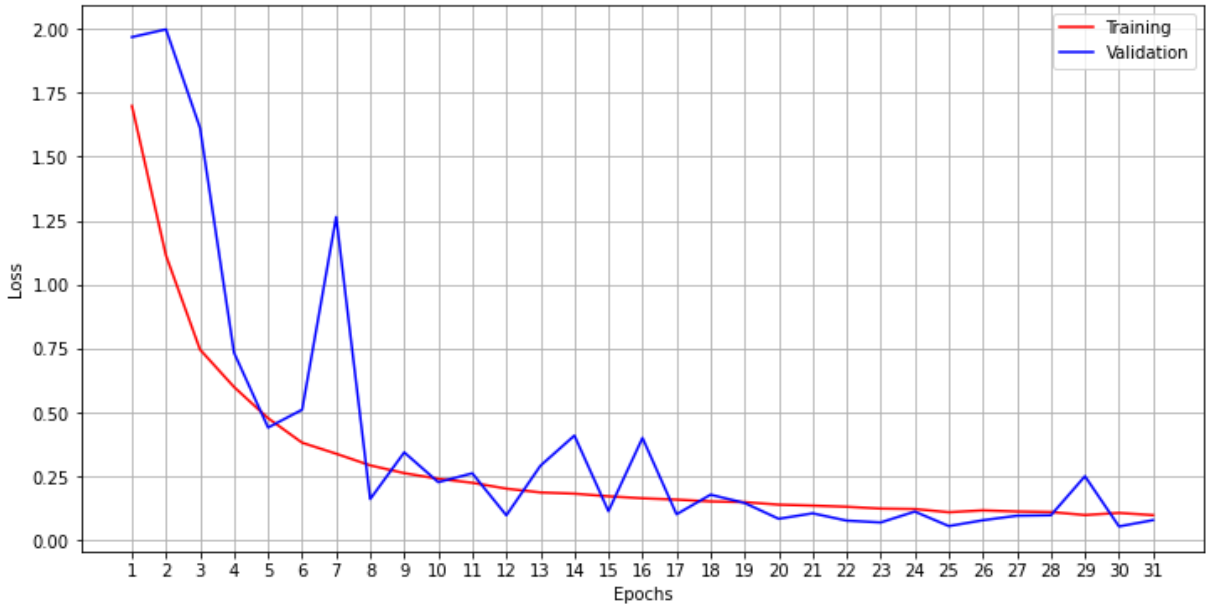


Figure 40 - Epoch/Loss Graph

According to the theory of model fitness approximation, [17] argues for an empirical approach of per model basis for selecting the most appropriate epoch that approximates the whole model fitness. For the previously discussed model, epoch number 23 was selected as it provided most appropriate evaluation metrics for accuracy and a balanced fitting empirical metric for an averagely fit model according to the reference graph shown in figure 41.

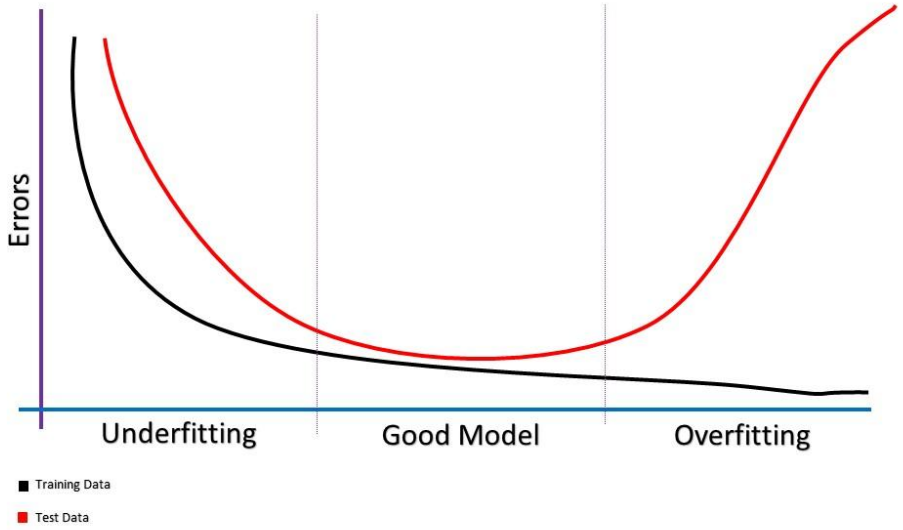


Figure 41 - Model Fitness Reference Graph

4.1.1. Architectures

Two of the six models are following the Alexnet standard approach, differing only in input size; 224x224 & 128x128 in order to test input dimensions on model size. The remaining models follow Mobilenet V1 approach with not only 224x224 & 128x128 input sizes but also different width multipliers; 1 & 0.25 each. 32x32 models were attempted to further reduce model size, however this introduced indistinguishable features and heavily lowered accuracy. This approach provided palpable metrics in order to make an educated decision on which architecture would provide the greatest leverage point towards efficiency while forgoing as little as possible in accuracy.

Following is our Alexnet 224x224 sequential model summary, as referenced by the seminal Alexnet paper. [3]

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
batch_normalization (BatchNo (1, 224, 224, 3)          12
_________________________________________________________________
conv2d (Conv2D)              (1, 222, 222, 32)         896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (1, 111, 111, 32)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (1, 109, 109, 64)         18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (1, 54, 54, 64)           0
_________________________________________________________________
conv2d_2 (Conv2D)            (1, 52, 52, 128)          73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (1, 26, 26, 128)          0
_________________________________________________________________
flatten (Flatten)            (1, 86528)                0
_________________________________________________________________
dense (Dense)                (1, 256)                  22151424
_________________________________________________________________
dense_1 (Dense)              (1, 12)                   3084
=================================================================
Total params: 22,247,768
Trainable params: 22,247,762
Non-trainable params: 6
_________________________________________________________________
```

Figure 42 - Alexnet Model Summary

4.1.2. Dataset

The dataset used is from PlantVillage, where our models were trained on 22,680 images, which were approximately equally split between the 12 classes to avoid accuracy paradox. [4] During our model training phase, an unequally divided dataset was used, this introduced us to the theory of accuracy paradox. An example would be a 2 class based dataset, where the first class is of apples and the other is of oranges; with the apples class having more training images than the oranges class. The model would create a function that describes that it is most probable for inputs to belong to the apples class and would reach the same conclusion when introduced to the evaluation inputs.

After applying a balanced dataset, the training images had image augments introduced to them by random shearing, zooming, rotation, dimensional shifts and horizontal flips. Which was forced to randomise at every epoch; this allows for variance in terms of accommodating for different image conditions. [5]

Our design forced an augmentation operation on the model at every new 12 hour session. At each 12 hour runtime, the previous model would be loaded, followed by the loading of the dataset and the augmentation operation. This provided a very high level of training variance that deep neural networks fundamentally require.

A 80:20 split for training and validation was forced, this allowed for 5,659 images used for validation during training. Where 3 images per class were left unintroduced to the model during the training phase, this allows for 36 images to be used for evaluation.

Following are the 12 classes that our models can classify and infer:

Table 1 - Plant Disease Model Classes

Apple	Corn	Tomato
Healthy	Healthy	Healthy
Scab	Cercospora leaf spot	Mosaic virus
Rust	Rust	Yellow leaf curl virus
Black rot	Blight	Early blight

4.1.3. Metrics

Aggregated metrics of the chosen 6 models are presented and discussed to provide contrast between them. The platforms are both Google cloud servers and RaspberryPi Zero with the models executed via Tensorflow lite. Moreover, the evaluation was done on the same input image across all models. All models present an accuracy of 98~99%. Time metrics are calculated based on an average of 3 runs; due to the fact of variance in the scheduler. The total time is the time of inferring plus the time to load the parameters, model and input image.

Table 2 - Model Metrics

Model	Alexnet		Mobilenet			
Input size(Width multiplier)	224x224	128x128	224x224(1)	224x224(0.25)	128x128(1)	128x128(0.25)
#Parameters	22,247,768	6,519,128	3,241,164	221,628	3,241,164	221,628
Parameters' size	85MB	25MB	13MB	1MB	13MB	1MB
Largest #parameters in same layer	22,151,424	6,422,784	1,048,576	65,536	1,048,576	65,536
Cloud infer time	0.386	0.373	0.824	0.798	0.773	0.754
Cloud total time	5.536	4.908	10.690	6.784	8.735	6.491
RPI infer time	6.996	2.304	8.431	0.840	3.186	0.266
RPI total time	7.204	2.496	8.642	1.051	3.381	0.462

It can be deduced that Alexnet models have considerably larger parameter size than Mobilenet models with little accuracy loss. Moreover, Mobilenet distributes the convolution operation by utilizing depthwise and pointwise convolutions which allows for smaller largest #parameters being in the same layer. Pinpointing the fact that Mobilenet 224x224(0.25) and 128x128(0.25) share similar metrics except the ones relating to execution times on RPi; this is due to the fact that the first layers have different image sizes as input. This allows for slightly faster execution times for the smaller input. The 1MB models open doors for further weight pruning and fixed point quantization which can further reduce the parameter size. Moreover, this smaller size will allow for on-fabric implementations which will introduce objectively fast and more efficient implementations.

4.1.4. Comparison

Comparing our implementation to other attempts to tackle similar applications provides feasibility.

Table 3 - Model Comparisons

Model	Mobilenet		Mobilenet [1]	Custom [2]
Top 1 Accuracy	98%	98%	89%	88.7%
Dataset Size	22,680	22,680	6,970	3,663
#Classes	12	12	6	Binary
Input size(Width multiplier)	224x224(0.25)	128x128(0.25)	224x224(0.5)	64x64
#Parameters	221,628	221,628	1.3 million	45,281
Parameters' size	1MB	1MB	5.46 MB	45K
Cloud infer time	0.798	0.754		
Cloud total time	6.784	6.491		
RPI infer time	0.840	0.266	Mobile 0.406	GPU Tesla -
RPI total time	1.051	0.462		

As it is an emergent application in the field of precision agriculture, the lack of abundant applications that carry out similar functionality is lacking. It can be concluded that compared to the current edge, our memory and storage footprint models find balance between accuracy and efficiency.

4.2. Custom CNN Implementation

After having trained 36 models that accomplished the feat of plant disease detection and having closely evaluated 6 of them, our choice was the MobileNet V1, with 128 resolution by 0.25 width multiplier, as it provided minimal loss in accuracy and an impactful reduction in model size. By using the TensorFlow high level libraries, our MobileNet model can be given image inputs of the 12 classes of plants and would output a correct classification with high accuracy, an example of an input of a healthy apple plant is shown below.

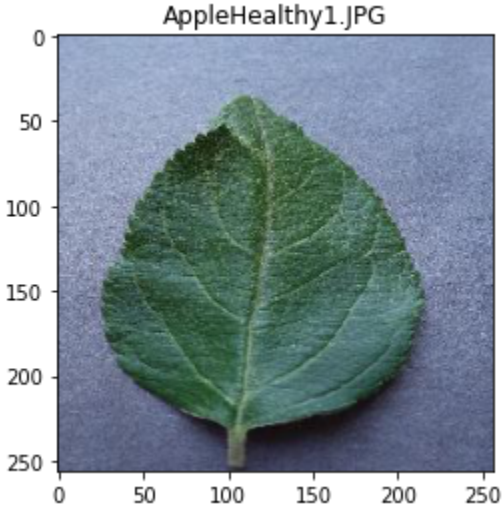


Figure 43 - Input to the Model

Following loading the input image, TensorFlow exposes a function inside its model class called predict, this function when given the input image as argument, would apply the forward pass of the parent model class and output a prediction as shown in figure 44.

```
Following is the prediction:
Apple___healthy : 0.99827945 %
```

Figure 43 - Output of the Model

TensorFlow is elegant and useful in many applications and for debugging, however it abstracts alot of its functionalities for the sake of simplicity. In the case of our design, it was essential to carry out the complex computations on the FPGA for acceleration, however the robust approach of TensorFlow proved indivisible. Thus a custom approach that aims to understand the optimized implementations of the TensorFlow functions and reconstruct them without the use of any of the TensorFlow libraries was undertaken.

In order to begin tackling such a problem, we first devised a list of the building blocks that are used to create the MobileNet V1 architecture [15], shown in the below figure are the building blocks and their counterpart TensorFlow implementations' documentation.

1.	Conv2D	tf.nn.conv2d | TensorFlow Core v2.4.1
2.	Depthwise Conv2D	tf.nn.depthwise_conv2d | TensorFlow Core v2.4.1
3.	BatchNorm	tf.nn.batch_normalization | TensorFlow Core v2.4.1
4.	ReLU	tf.nn.relu | TensorFlow Core v2.4.1
5.	ZeroPadding2D	tf.pad | TensorFlow Core v2.4.1
6.	GlobalAveragePooling2D	tf.nn.avg_pool2d | TensorFlow Core v2.4.1
7.	Reshape	tf.reshape | TensorFlow Core v2.4.1
8.	SoftMax	tf.nn.softmax | TensorFlow Core v2.4.1

Figure 44 - Building Block Documentation

To allow for complete project documentation, the methodology for reconstructing the vectorized Conv2D will be closely reviewed as an example for the methodology applied. In the case of Conv2D reconstruction we were given a high level algorithm which we must produce a low level implementation from. Shown is the TensorFlow documentation for their Conv2D implementation.

Given an input tensor of shape `batch_shape + [in_height, in_width, in_channels]` and a filter / kernel tensor of shape `[filter_height, filter_width, in_channels, out_channels]`, this op performs the following:

1. Flattens the filter to a 2-D matrix with shape `[filter_height * filter_width * in_channels, output_channels]`.
2. Extracts image patches from the input tensor to form a *virtual* tensor of shape `[batch, out_height, out_width, filter_height * filter_width * in_channels]`.
3. For each patch, right-multiplies the filter matrix and the image patch vector.

Figure 44 - TensorFlow Conv2D Implementation Documentation

From this description, we have the knowledge that TensorFlow leverages the use of vectorization via the use of patch extraction to achieve their Conv2D implementation. With the aid of our continually growing Google Cloud Model Suite, we created a Debug Suite that was essential in component testing. Following is the Conv2D process over a simple example image.

Given a 3 channel image with dimensions 5x4, shown in figure 45. Highlighted in green is an example of a 3x3 kernel situated at the start of the input image.

```
[0,1,2,3]
[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]
[36,37,38,39]

[40,41,42,43]
[44,45,46,47]
[48,49,50,51]
[52,53,54,55]
[56,57,58,59]
```

Figure 45 - Image Example

In the standard spatial convolution, the 3x3 kernel in the green position would execute an element wise multiplication followed by an accumulation operation, and then stride until the whole image is convolved with the kernel. However, in the case of vectorization, the multiply accumulation operation is not applied at every stride, a patch extraction operation is. The kernel would stride normally, but it would gather the input components in such a way that the convolution operation is equivalent to a matrix multiplication operation. Shown in figure 46 is the output of our custom implementation of the vectorization function.

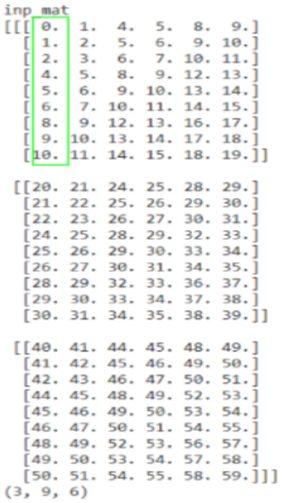


Figure 46 - Vectorized Image Example

Highlighted in green in figure 46 is the vectorization operation applied to the green section of figure 45. As can be deduced, the vectorized representation of the input image carries increased storage footprint, yet leverages increased performance. Shown in table 4, a comparison between the standard spatial convolution, our vectorized implementation and the TensorFlow implementation.

Table 4 - Conv2D Implementation Metrics

Metrics	Spatial	Vectorized	TF
Exec time (126x126x32)	4.9	0.8	0.28

Our vectorized implementation leverages similar execution time over the same input compared to TensorFlow's implementation as compared to the spatial approach.

This methodology was executed for each of the building blocks shown in figure 44 and collected in a file aptly named Low Level Functions. Shown in table 5 are the functions housed inside the LL Functions file.

Table 5 - Low Level Functions

Input Reshape	Kernel Reshape	Pad	Batch Norm	ReLU	SoftMax	ZeroPad	Load Weights	Load File	Display

These low level functions are used as tools to implement the MobileNet V1 architecture, these LL functions are included in the file named High Level Functions whereby the LL functions are used to implement higher level functionalities that more naturally implement the model architecture, shown in table 6 are the High Level Functions and the Low Level Functions that they utilize.

Table 6 - High Level Functions

Conv Block	Conv2D	Conv2D Parallel Reshape
Load Weights, Conv2D, Batch Norm, ReLU, ZeroPad	Pad, Conv2D Parallel Reshape, Numpy.MatMul	Input Reshape, Kernel Reshape

This high level function approach of having the Conv Block component represent the model architecture is influenced by the MobileNet V1 white paper. [15] With this approach we were able to successfully reconstruct a fully custom MobileNet V1 forward pass implementation that does not rely on any TensorFlow libraries, an implementation that we can control and divide according to our requirements.

4.3. System Functional Simulation

After completion of our custom CNN implementation, focus was drawn into the introduction of the hardware acceleration component of the system. In order to execute component testing, we implemented the algorithm that is being utilized to design our custom Single Purpose Processor in Python, shown in figure 47.

```
# VHDL code emulation
def mat_mult(temp_ker_mat, temp_inp_mat, ker_row, ker_col, inp_row, inp_col):

  output = np.zeros([ker_row, inp_col])

  for i in range(ker_row):
    for j in range(inp_col):
      for x in range(ker_col):
        output[i][j] += (temp_ker_mat[i][x] * temp_inp_mat[x][j])

  return output
```

Figure 47 - Matrix Multiplication Algorithm

The mat_mult function shown in figure 47, very closely emulates our SPP and can be replaced by the Numpy MatMul function to achieve correct matrix multiplication. But instead of directly replacing the function, it was decided that by using Python threading libraries we would more accurately reach a simulation of the whole system as the concurrent components interface with each other. We introduced three threads to the simulation; a UI thread which would emulate a user inputting an image, an API thread which would receive the input image and apply the model upon it and lastly the FPGA thread which would be utilized by the API to calculate the matrix multiplications. In order for these three components to realistically interface with one another, we utilized the use of pointer arrays to emulate the function of a shared memory. A shared memory named Socket Ref was situated between the UI thread and API to emulate an open file descriptor (socket) between them. And another shared memory between the API and FPGA to emulate the master to slave peripheral interfacing. On the specific stage that a matrix multiplication operation is required, the API would assign values to the shared memory that correspond to the Mat_Mult's inputs, followed by issuing it to begin its Mat_Mult algorithm. Upon completion the FPGA would assign the result onto the shared memory for the API to obtain. This approach of threading introduced a rudimentary implementation of the peripheral datasheet for interfacing, this approach will be refined and explored in the custom peripheral section of the report. This interaction between the API and the FPGA thread is repeated until all layers of the model are accounted for. After completion the API would return the inferred output to the UI thread via their shared memory component.

In order to properly achieve MobileNet V1 architecture we designed block diagrams that outline the system execution pipeline. Shown in figure 48 is the interfacing between the UI thread and

the socket reference. In practice, the socket reference is an array pointer that aims to emulate a socket communication between a proper user interface and our emulated API that handles the backend operations and interfaces with the FPGA thread.

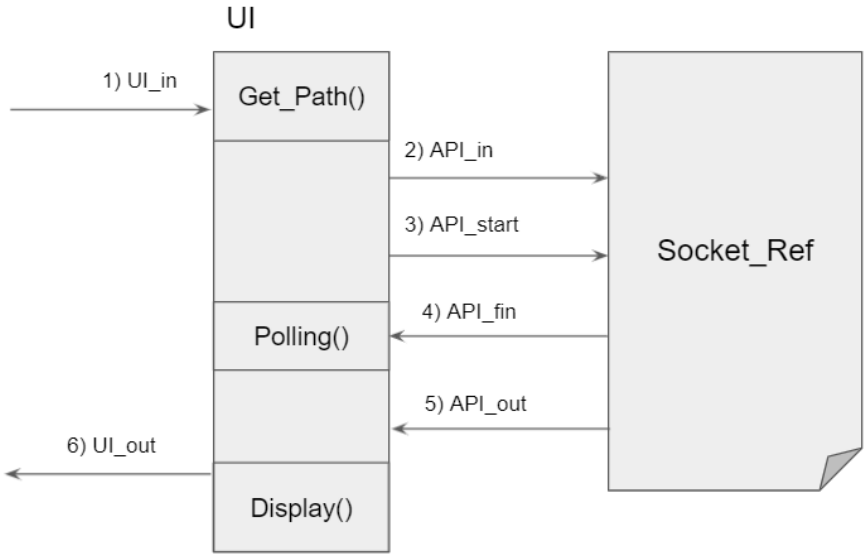


Figure 48 - UI & Socket interfacing

The UI thread gets an image path as input and assigns its value to API_in. Then asserts API_start. The UI thread remains in idle mode polling the API_fin value. When API_fin is asserted, the UI takes the value assigned to the API_out and passes it through the Display function which handles displaying the inferred value similar to the output shown in figure 43.

Figure 49 entails the interaction between the socket reference and the API.

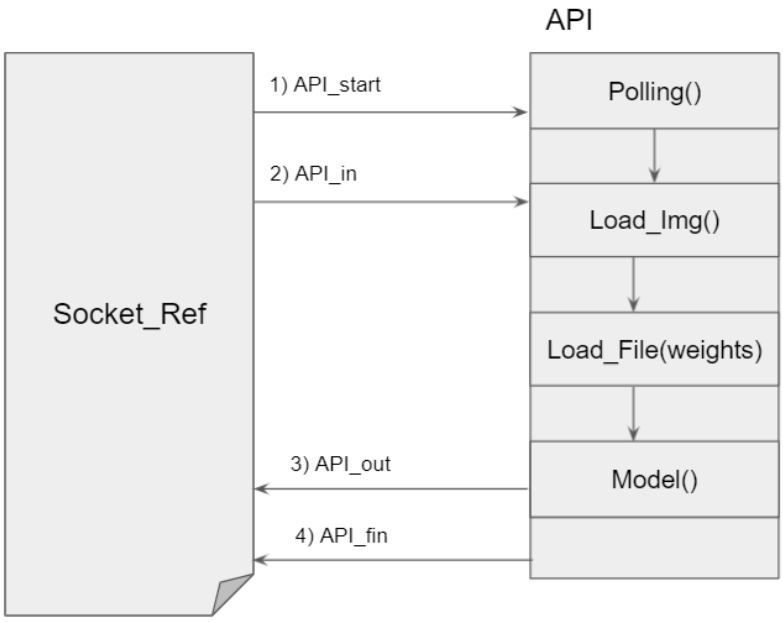


Figure 49 - Socket & API interfacing

Having polled an asserted API_start, the API takes in the value of API_in and uses it to load the input image into memory, followed by loading the model weights and calling the Model function. Upon the return of the Model function, the return value is assigned to API_out and API_fin is asserted.

Figure 50 discloses the functionality of the Model Function.

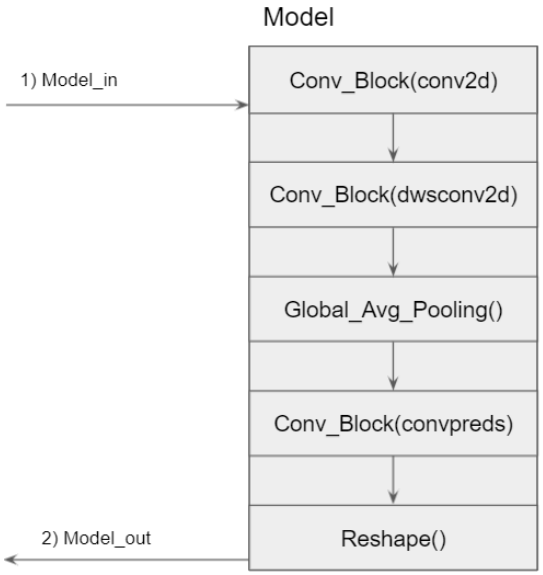


Figure 50 - MobileNet V1 Model

This function block exposes the high level functions called in order to achieve the MobileNet V1 architecture. Having the input image introduced to a convolutional block of type Conv2D, followed by its output inserted into 13 consecutive convolutional blocks of type DWS Conv2D. Following that, having its output introduced to a Global Average Pooling layer and finally a final convolutional block that begins the process of reshaping to meet the 12 class model output result.

Diving deeper into the implementation of the MobileNet V1 architecture, the high level function Conv_Block takes in several arguments in order to achieve similar functionality compared to TensorFlow's MobileNet V1 implementation. [18] Figure 51 shows its inner workings.

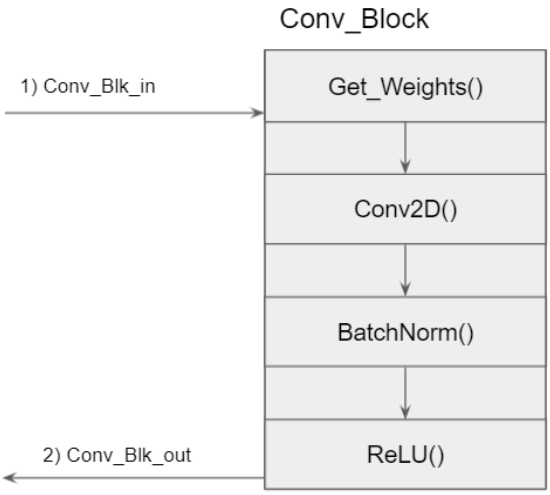


Figure 51 - Convolutional Block Function

For each block, we get its respective weights from memory and apply the Conv2D function, followed by a batch normalization and a ReLU operation. Thus achieving the exact convolutional block as disclosed in the MobileNet V1 white paper. [15]

An important approach that was applied in our Conv2D implementation is the fact that by leveraging the vectorization method we were able to unify both the standard Conv2D and the depth-wise Conv2D by treating them both indiscriminately as depth-wise convolutions and accounting for distinctions at the end of the operation. With the aid of our debugging tools and the advanced understanding of the problem that we have developed, we were able to successfully lift up the standard Conv2D to the same level of parallelism that depth-wise Conv2D bolsters.

Through understanding of the low level functionality of both types of convolutions, we were able to infer that there are only two differing factors that can be accounted for between Conv2D and DW Conv2D. These two factors can be accounted for at the end of our Conv2D implementation. Figure 52 shows the Conv2D function.

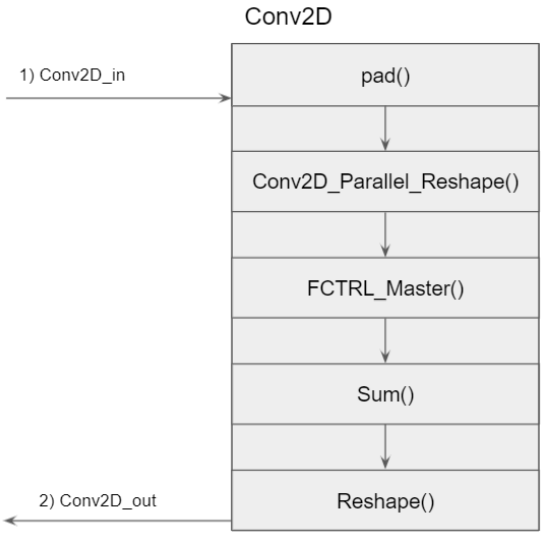


Figure 52 - Unified Convolutional Function

In both cases, a padding operation is executed for each convolutional block. The type of padding applied is specified based on the MobileNet V1 architecture. [15] The Conv2D parallel reshape function handles the reshaping of the kernel and input spatial tensors into their vectorized matrix representations. This function applies reshaping in a manner equivalent to the DW Conv2D; where each kernel is applied to only one channel. This process creates massive parallelism for the standard Conv2D. An important challenge was introduced at this point. In the DW Conv2D operation, there exists two clauses: 1) Must have only one kernel. 2) The kernel's depth must be equal to that of the input depth. Thus, only convolutions over the depth occur, as explored previously, this design introduces parallelism for multiple convolutions concurrently applied. In the case of standard Conv2D, DW Conv2D's second clause is enforced. However, the first clause is not, the standard Conv2D can have any number of kernels applied to the input. Herein lies the practical challenge. Through our debugging suite, we were able to overcome this challenge by reshaping the kernel in such a way such that the output matrix is three dimensional tensor, and for each depth dimension the values of the kernel matrix are flattened row-wise and grouped by the input depth they interact with. Figure 53 entails the kernel reshape method.

```
[0,1,2]
[3,4,5]
[6,7,8]

[9,10,11]
[12,13,14]
[15,16,17]

[18,19,20]
[21,22,23]
[24,25,26]
```

```
ker_mat
[[[ 0.  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.]]]
(3, 1, 9)
```

Figure 53 - Kernel Reshape

The aptly named flow control master (FCTRL_Master) controls/regulates the flow of data between the API and the FPGA in order to input the kernel and input matrices and output the matrix multiplication result. Until now both the Conv2D and DW Conv2D are treated equally, however when applying the standard Conv2D, an element wise summation occurs over the depth of the output matrix in order to emulate the standard convolution operation. As for DW Conv2D the output matrix is in no need of change.

Finally, by functionally simulating the FCTRL components we can realistically design a protocol for proper API to FPGA interfacing. Starting with the flow control master function shown in figure 54.

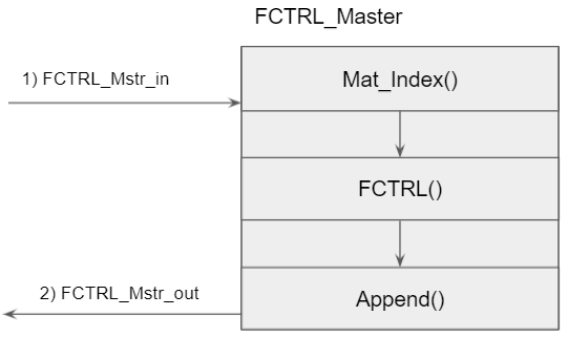


Figure 54 - Flow Control Master

The FCTRL master function receives the multidimensional, vectorized and reshaped input and kernel matrices. In order to realistically approach the problem of flow control, our design only supports the matrix multiplication of two matrices, where each matrix is only two dimensional. Thus, the two input matrices have their depth values indexed via the Matrix Index function.

The FCTRL function takes the two dimensional matrices and writes and reads on the shared memory that is situated between the API and the FPGA, commanding the SPP that is synthesized on the FPGA to start operation. After the FPGA completes the matrix multiplication operation, the output value is appended. These three functions of the FCTRL Master remain in a loop until the depth of the inputs are covered. Then returns the fully appended output.

The way the FCTRL function interacts with the shared memory component is simple, it assigns the input values to the shared memory and asserts FPGA_start then polls for FPGA_fin and on assertion returns the FPGA_out value. As shown in figure 55.

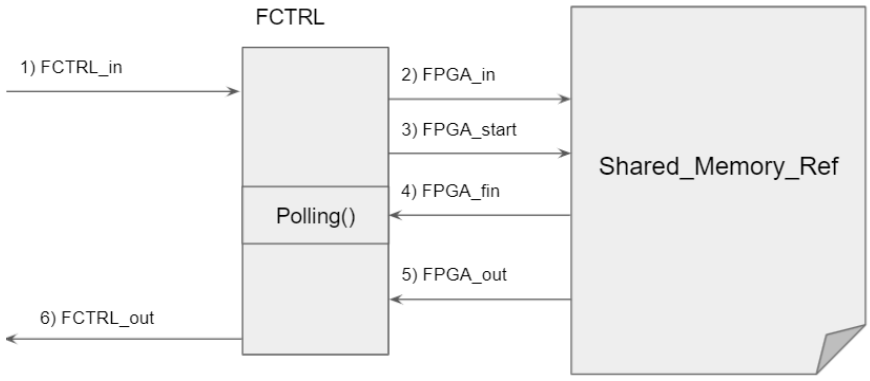


Figure 55 - Flow Control & Shared Memory Interfacing

On the other side, as shown in figure 56, the FPGA is polling the FPGA_start and on assertion, takes the values in FPGA_in and begins its Matrix Multiplication routine. On Completion, the matrix multiplication output is assigned as well as the FPGA_fin asserted.

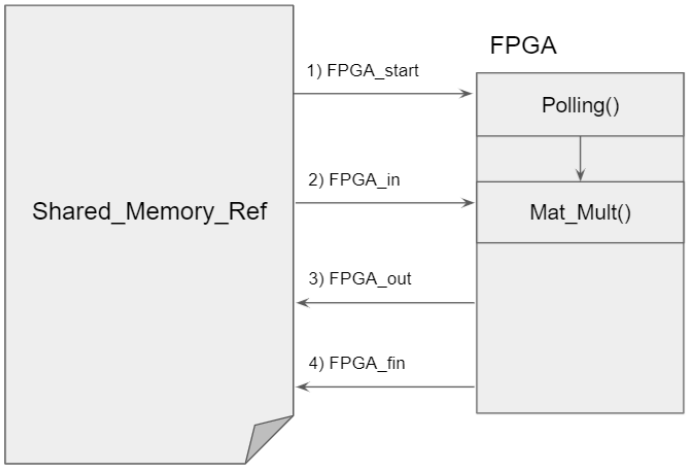


Figure 56 - Shared Memory & FPGA Interfacing

By following the thread of execution, one can understand the full functionality of our designed system. This design was used to implement the system functional simulation. Following are the aggregate results of the functional simulation running on Intel(R) Core(TM) i7-8700, Google Cloud and the Arm Cortex A9 of the DE10 Standard FPGA.

Table 7 - Functional Simulation Metrics

Device	Intel i7-8700	Google Cloud	Arm Cortex A9
Accuracy	99.7 %	99.7 %	99.7 %
Total Time (sec)	13.7	27.4	309.8

Given the same inputs and having the average of 3 attempts averaged returns the following table. The Intel and Google Cloud CPU architectures follow the x86 CPU architecture. However, Arm Cortex employs its own reduced instruction architectures to enable low power consumption on embedded devices like the DE10 Standard board. This explains the large difference in the time taken to run the same source code on the Arm Cortex device.

4.4. Custom Single Purpose Processor

A matrix is a powerful tool for storing and manipulating data in many fields. Linear algebra produces many operations of dealing with matrices like addition, subtraction, multiplying, transpose…, etc. Applying these operations is helpful in many fields like statistics, applied mathematics, economics, artificial intelligence, physics, and so many other fields. What makes the matrix special is that storing different numbers with some sort of indices. There are two types of the indices: row indices and column indices. Therefore, that makes the access to specific data in the matrix easy. Addition and subtraction would also be so easy to apply in this form of data. The concern is in matrix multiplication, which is needed in multiplying image matrix with its kernel matrix. Matrix multiplication is different from applying and has to have some conditions to apply:

- If two matrices are multiplied: Matrix1 has dimensions (m1, n1), matrix2 has dimensions (m2, n2), m is the number of rows and n is the number of columns, then n1 must be equal to m1.
- If two matrices are multiplied: Matrix1 has dimensions (m1, n1), matrix2 has dimensions (m2, n2), and the output matrix has dimensions (m3, n3), then m3 must be equal to m1 and n3 must be equal to n2.

Matrix multiplication works by a unique process. The first row elements from the first matrix are multiplied by the first column' elements, and the values from the multiplying are added together and stored in the first row and first column of the output matrix, then same process for the first row's elements with the second column's elements, …..., and so on. Given that A, B are matrices with dimensions (m, n) and (n, p) respectively, then C = A x B has dimensions (m, p) like it is shown here:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & a_{11}b_{12} + \cdots + a_{1n}b_{n2} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ a_{21}b_{11} + \cdots + a_{2n}b_{n1} & a_{21}b_{12} + \cdots + a_{2n}b_{n2} & \cdots & a_{21}b_{1p} + \cdots + a_{2n}b_{np} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & a_{m1}b_{12} + \cdots + a_{mn}b_{n2} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{pmatrix}$$

The algorithms that have been designed to apply multiplication between two matrices are many and different. However, before exploring the algorithms, there are two types of matrices that have to be defined first:

- Square Matrices: Square matrices are matrices whose dimensions are equal. For instance: Matrix a with dimensions (m, n), m is equal to n.

- Rectangular Matrices: Rectangular matrices are matrices whose dimensions do not have to be equal. For instance: Matrix a with dimensions (m, n), m is equal to n, or m is not equal to n.

These two types define the nature of the algorithm that applies matrix multiplication. For square matrix multiplication, the design of the algorithm is easy given that the number of rows and number of columns are the same. Given that algorithm which applies the multiplication of two square matrices:

This algorithm multiplies A and B and stores the result in C. A and B have dimensions of n x n.

Input parameters: A, B

Output parameters: C

```
Matrix_product_square(A, B, C){
    n = A.last
    for i = 0 to n
        for j = 0 to n {
            C[i][j] = 0
            for k = 0 to n
                C[i][j] = C[i][j] + A[i][k] * B[k][j]
        }
}
```

Time complexity: $O(n^3)$.

Auxiliary space: $O(n^2)$

Time complexity can be reduced if Strassen's matrix multiplication is used. However, the image matrix and kernel matrix are rectangular. Therefore, the *Matrix_product_square* algorithm would not be useful. Another algorithm to apply matrix multiplication on rectangular matrices has to be available. Given that algorithm which applies the multiplication of two square matrices:

This algorithm multiplies A and B and stores the result in C. A and B have dimensions of m1 x m2 and n1 x n2, respectively.

Input parameters: A, B

Output parameters: C

```
Matrix_product_rectangular(A, B, C, m1, m2, n1, n2){

    for i = 0 to m1 - 1
        for j = 0 to n2 - 1 {
            C[i][j] = 0
            for k = 0 to m2 - 1
                C[i][j] = C[i][j] + A[i][k] * B[k][j]
        }
}
```

Time complexity: $O(n^3)$.

Auxiliary space: O (m1 * n2)

This algorithm enables the design of a digital circuit that applies rectangular matrix multiplication in RTL. The RTL design will be in FSMD (data path + FSM) architecture. The data path stores the data of the algorithm, applies the arithmetic operations and passes the conditions of the for loops to the controller (FSM).

On the other hand, FSM controls the flow of the data in the algorithms and writes the result in the output ram. The architecture takes:

- mat1: The output of the ram/rom that stores the first matrix needs to be multiplied.
- Mat2: The output of the ram/rom that stores the second matrix needs to be multiplied.
- m1: The number of rows in the first matrix.
- m2: The number of columns in the first matrix.
- n1: The number of rows in the second matrix.
- n2: The number of columns in the first matrix.
- res_in: The output of the result ram.
- start: A signal to initiate the process of multiplying matrices.

The output signals:

- i_out, j_out, k_out : The addresses of the given matrices and the output matrix (I, j, k), as they are specified in the algorithm.
- res_out: The value of an element in the output matrix.
- wr: Signal to write res_out in the output ram.
- finish: Signal to inform that the process has been terminated.

This figure shows the architecture of the data path.

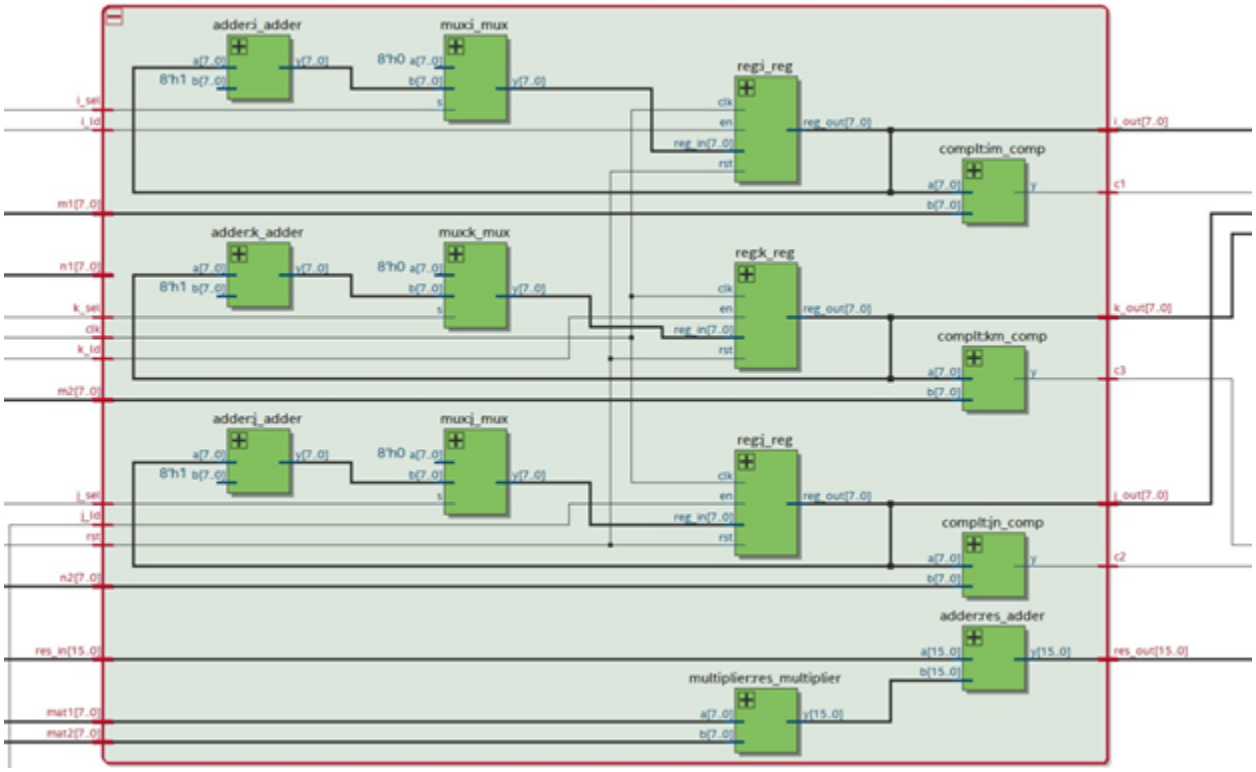


Figure [1].

The data path architecture' inputs are:

- i_sel: Signal comes from the controller to control the selector signal of i_mux.
- i_ld: Signal comes from the controller to control the enable signal of i_reg.
- m1: The number of rows of the first matrix.
- n1: The number of rows of the second matrix.
- k_sel: Signal comes from the controller to control the selector signal of k_mux.
- k_ld: Signal comes from the controller to control the enable signal of k_reg.
- m2: The number of columns of the second matrix.

- j_sel: Signal comes from the controller to control the selector signal of j_mux.
- j_ld: Signal comes from the controller to control the enable signal of j_reg.
- n2: The number of columns of the second matrix.
- res_in: An element of the output matrix.
- mat1: An element of the first matrix.
- mat2: An element of the second matrix.

The data path architecture' outputs are:

- i_out: The signal of address i.
- c1: Signal of comparing i with m1.
- k_out: The signal of address k.
- c3: Signal of comparing k with m2.
- j_out: The signal of address j.
- c2: Signal of comparing j with n2.
- res_out: The element to be stored in the output ram.

The data path architecture' components are:

- i_adder: It is an adder to apply addition on the output signal of i_reg and one.
- i_mux: It is a multiplexer to select the input of i_reg either it is the signal that comes from i_adder or zero. The selector of i_mux is i_sel.
- i_reg: It is a register to store the signal that comes from i_mux. The enable signal of i_reg is i_ld.

- im_comp: It is a comparator that outputs one if the signal coming from i_reg is less than m1 and 0; otherwise. c1 is the signal that comes from im_comp.

- k_adder: It is an adder to apply addition on the output signal of k_reg and one.

- k_mux: It is a multiplexer to select the input of k_reg either it is the signal that comes from k_adder or zero. The selector of k_mux is k_sel.

- k_reg: It is a register to store the signal that comes from k_mux. The enable signal of k_reg is k_ld.

- km_comp: It is a comparator that outputs one if the signal coming from k_reg is less than m2 and 0; otherwise. c3 is the signal that comes from km_comp.

- j_adder: It is an adder to apply addition on the output signal of j_reg and one.

- j_mux: It is a multiplexer to select the input of j_reg either it is the signal that comes from j_adder or zero. The selector of j_mux is j_sel.

- j_reg: It is a register to store the signal that comes from j_mux. The enable signal of j_reg is j_ld.

- jn_comp: It is a comparator that outputs one if the signal coming from j_reg is less than n2 and 0; otherwise. c2 is the signal that comes from jn_comp.

- res_multiplier: It is a multiplier to apply multiplication on the mat1 and mat2 signals.

- res_adder: It is an adder to apply addition on the output signal of res_multiplier and res_in.

The idea of the data path is to apply the arithmetic operations of the algorithms, send the conditions signals to the controller, receive the control signals of the registers and multiplexers from the controller, and generate the elements of the output matrix with their addresses.

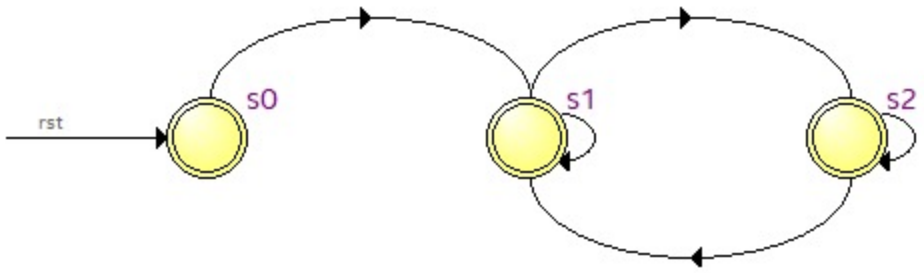


Figure [2].

As it is shown in this figure [2], the FSM contains three states. The first state s0, generally, spares one clock period of time and transits to s1 unconditionally. State s1 initializes the data with their initial values to be prepared before initiating the loops in s2. State s2 implements the for loops in the algorithm given the conditions signals from the data path.

	Source State	Destination State	Condition
1	s0	s1	
2	s1	s2	(start)
3	s1	s1	(!start)
4	s2	s2	(c1)
5	s2	s1	(!c1)

Figure [3].

This table shows the transitions from one state to another. State s0 transits to s1 without any conditions. State s1 would transit to s1 if the signal starts enabled; otherwise, it remains in state s1. State s2 remains in s2 as long as I is less than m1 (signal c1 that comes from the data path); otherwise, the algorithm terminates and returns back to state s1, waiting for another start signal to start the process again.

The controller architecture' outputs are:

- i_sel: Signal comes from the controller to control the selector signal of i_mux.
- i_ld: Signal comes from the controller to control the enable signal of i_reg.
- k_sel: Signal comes from the controller to control the selector signal of k_mux.
- k_ld: Signal comes from the controller to control the enable signal of k_reg.
- j_sel: Signal comes from the controller to control the selector signal of j_mux.
- j_ld: Signal comes from the controller to control the enable signal of j_reg.
- wr: Signal to write res_out in the output ram.

The controller architecture' inputs are:

- c1: Signal of comparing i with m1.
- c3: Signal of comparing k with m2.
- c2: Signal of comparing j with n2.
- start: Signal to begin the process of the algorithm.

The outputs of the controller differ given the conditions signals that come from the data path (c1, c2, c3).

In state s0:

i_sel = 0.

i_ld = 0.

k_sel = 0.

k_ld = 0.

j_sel = 0.

j_ld = 0.

wr = 0.

finish = 0

In state s0, nothing happens but transition to state s1.

In state s1:

i_sel = 0.

i_ld = 1.

k_sel = 0.

k_ld = 1.

j_sel = 0.

j_ld = 1.

wr = 0.

finish = 0

In-state s1, the registers i_reg, j_reg, k_reg are stored by a value equal to zero. i_sel, j_sel, and k_sel make their multiplexers output zero when they are equal to zero. In-state s2:

The truth table is implemented given the conditions of the loops.

CONTROL SIGNALS TRUTH TABLE

C1	C2	C3	wr	k_ld	k_sel	i_ld	i_sel	J_ld	J_sel
0	0	0	0	0	X	0	X	0	X
0	0	1	0	0	X	0	X	0	X
0	1	0	0	0	X	0	X	0	X
0	1	1	0	0	X	0	X	0	X
1	0	0	0	1	0	1	1	1	0
1	0	1	0	1	0	1	1	1	0
1	1	0	0	1	0	0	X	1	1
1	1	1	1	1	1	0	X	0	X

Figure [4].

As it is shown in the table:

- When c1 (i < m1) is not enabled, the algorithm terminates, and the FSM transits to s1. k_ld, i_ld, j_ld, and wr signals are not enabled. Finish signal will be enabled.

- When c1 (i < m1) is enabled and c2 (j < n2) is disabled, the algorithm is still in the first loop but the second loop terminates. Therefore, i = i + 1 (i_ld = 1, i_sel = 1), j = 0 (j_ld = 1, j_sel = 0), k =0 (k_ld = 1, k_sel = 0) and wr signal is disabled.

- When c1 (i < m1) is enabled, c2 (j < n2) is enabled and c3 (k < m2) is disabled, the algorithm is still in the first two loops but the third loop terminates. Therefore, j = j + 1 (j_ld = 1, j_sel = 1), k =0 (k_ld = 1, k_sel = 0) and wr and i_ld signals are disabled.

- When c1 (i < m1) is enabled, c2 (j < n2) is enabled and c3 (k < m2) is enabled, the algorithm is in the third loop. Therefore, k = k + 1 (k_ld = 1, k_sel = 1), wr = 1 and i_ld and j_ld signals are disabled.

The top level of the architecture should be like in this figure.

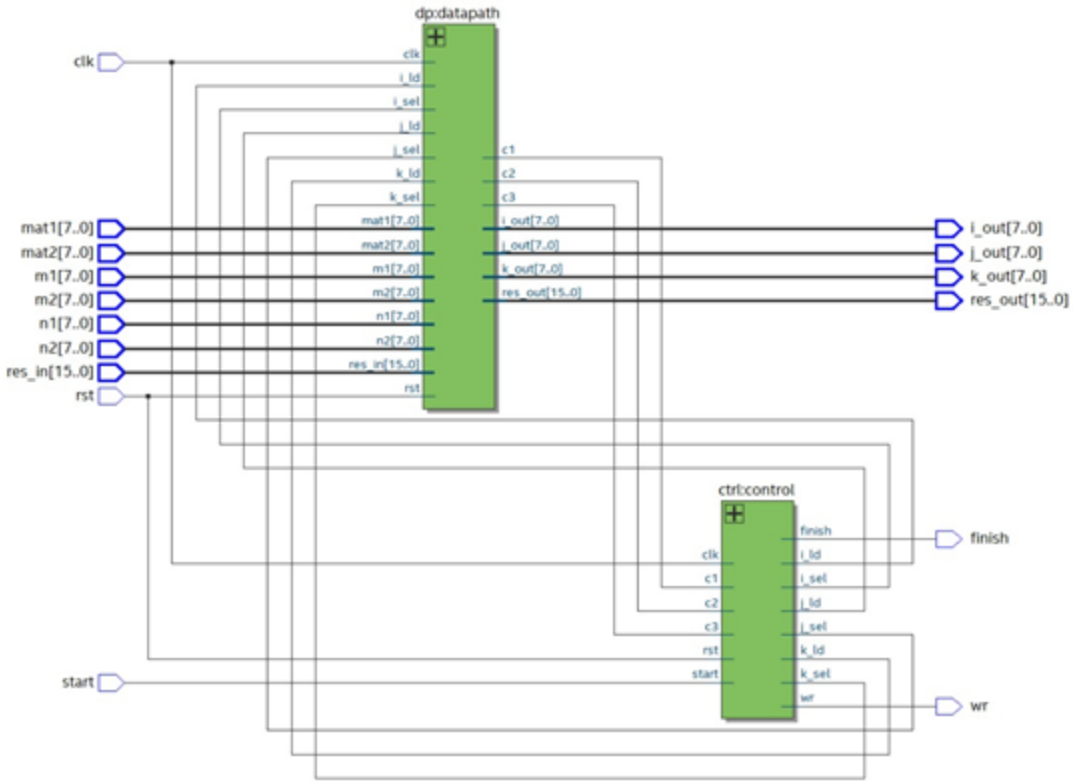


Figure [5].

As it is shown in the figure, the data path outputs the c1, c2, and c3 signals to the controller and takes i_ld, j_ld, k_ld, i_sel, j_sel, and k_sel from the controller. To implement this architecture on given matrices, the two matrices should be stored in ROMs / rams with the number of rows in the first ROM / RAM > m1 and the number of columns > m2, and the number of rows in the second RAM/ ROM > n1 and the number of columns > n2. The top-level apply the multiplication on the first m1 rows and m2 columns from the first ram and the first n1 rows and n2 columns from the second matrix. Then, it outputs the first m2 rows and n1 columns from the result matrix in a ram that takes the wr signal from the controller, res_out from the data path, i_out for the address of the row, and j_out for the address of the column. The first RAM / ROM that stores the first matrix takes i_out as the address for the row and k_out as the address of the column. The second RAM / ROM that stores the second matrix takes k_out as the address for the row and j_out as the address of the column.

Testing for a simple example, given a matrix A 2 x 3 and B 3 x 2 to be like that:

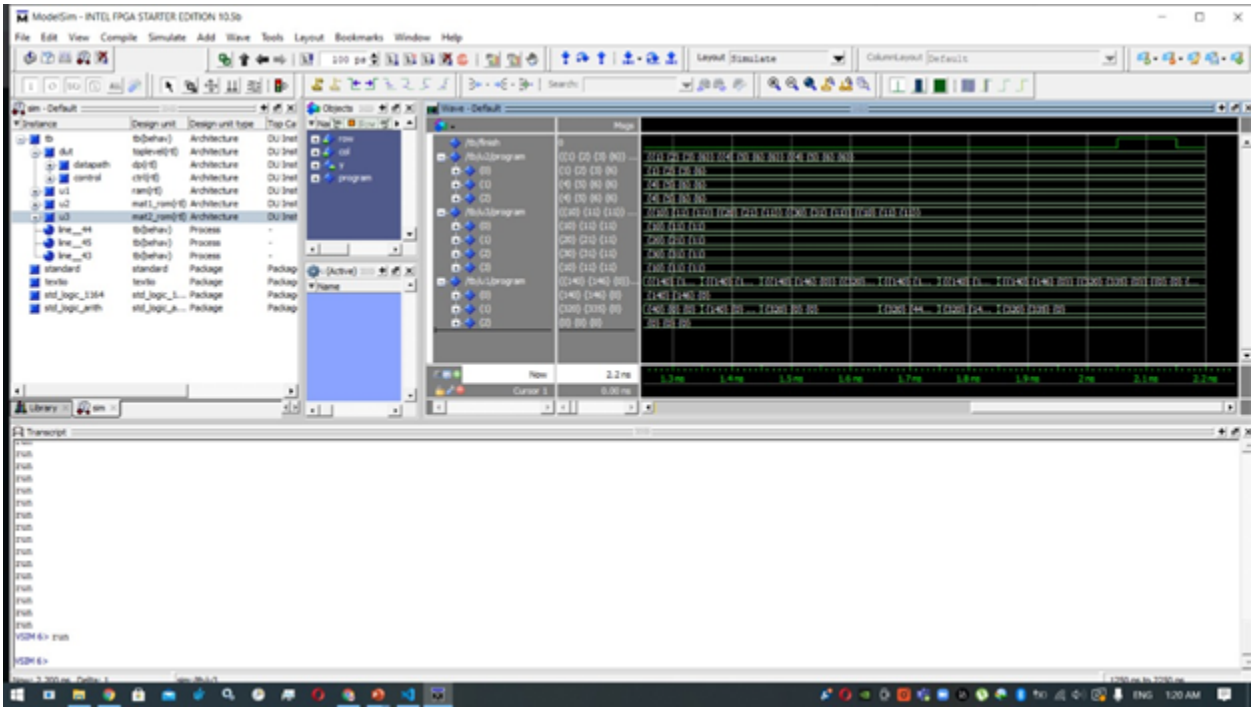
$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$B = \begin{bmatrix} 10 & 11 \\ 20 & 21 \\ 30 & 31 \end{bmatrix}$$

A matrix is stored in a ROM 3 x 4, which matrix A takes only the first two rows and first three columns.

B matrix is stored in a ROM 4 x 3, which matrix B takes only the first three rows and first two columns.

The simulation will be like in this figure.



The extra rows and columns in the two input ROMs are stored with arbitrary values. The three rows and two columns in the result matrix give the output matrix of multiplying the A matrix and B matrix.

$$result = \begin{bmatrix} 140 & 146 \\ 140 & 146 \\ 320 & 335 \end{bmatrix}$$

Certainly, the dimensions of the matrices are given to the top level. m1 = 2, m2 = 3, n1 = 3 and n2 = 3.

4.5. Custom Peripheral

Looking at the current state of the project, the custom CNN is running on the HPS, the custom matrix multiplication SPP is simulated and ready for synthesis on the FPGA and finally the Avalon Bus MM Interface can be introduced and an understanding of master & slave interfacing is added. Whereby, the only remaining step is wrapping the SPP around an interface level abstraction, to allow for the HPS & FPGA transreceving. With the knowledge of peripheral terminology; we want to create a slave peripheral, whereby it has two masters; the HPS and the FPGA. Following is the block diagram for the custom peripheral.

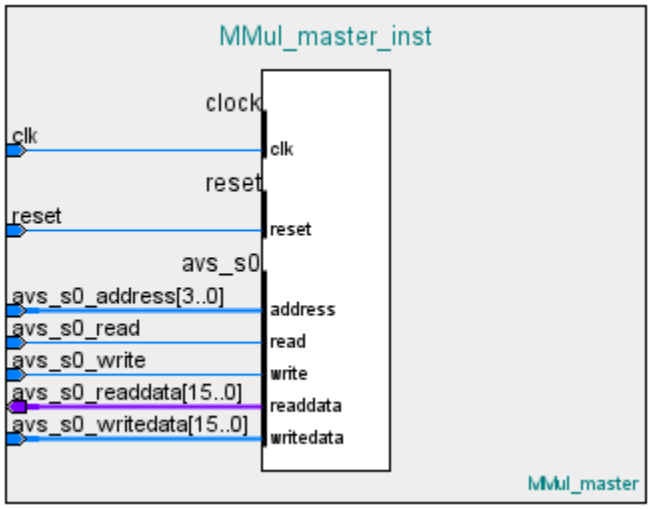


Figure 57 - Peripheral Block Diagram

The peripheral is set up in such a way that the avs_s0 IOs are controlled directly by the peripheral masters (HPS & FPGA). Whereby the masters can communicate with each other in a similar manner like in the case of the system functional simulation discussed previously. With the interfacing line available, a custom interfacing protocol was needed to streamline the HPS-FPGA interface.

Our design follows a 16 bit data word register width to allow for reduced processing footprint and minimal bus width. Following are the peripheral registers.

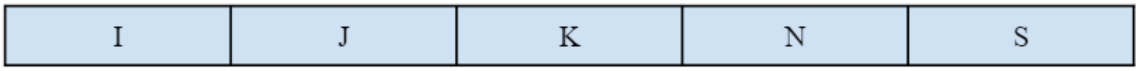


Figure 58 - Peripheral Registers

The functionalities of the I, J, K and N registers change depending on the value of the S register. Exploring the different values that the S register holds will shine a light in understanding the functionality of the peripheral. In short, the peripheral has six states of operation. Setting the S register to the corresponding hex value will utilize the I, J, K and N registers in different manners.

1) Loading Kernel Matrix (0x11)
2) Loading Input Matrix (0x21)
3) Matrix Multiplication Start (0x06)
4) Matrix Multiplication Finish (0x08)
5) Unloading Output Matrix (0x48)
6) Completion/Reset (0x0A)

Shown below is the structure of the S register.

Figure 59 - Peripheral S Register Structure

Having the S register in the kernel loading state assigned the value of 0x11 is designed in such a way so that the I and J registers will hold the Row and Column index values respectively. While having the N register hold the numeric value indexed at (I,J). Thus, the HPS component can load the two dimensional matrix onto the SPP memory. Similarly, in the input loading state with S register assigned 0x21, the I register is used for Row indexing and the K register is used for column indexing, with the N register holding the numeric value at (I,K). In practice, the system implementation loops at each value until both the kernel and input matrices are loaded into the shared memory.

X	X	X	X	X	X	X	X	X	ldout	ldinp	ldker	mfin	mstrt	ldfin	ldstrt
X	X	X	X	X	X	X	X	X	0	0	1	0	0	0	1

Figure 60 - Peripheral S Register During Kernel Loading

X	X	X	X	X	X	X	X	X	ldout	ldinp	ldker	mfin	mstrt	ldfin	ldstrt
X	X	X	X	X	X	X	X	X	0	1	0	0	0	0	1

Figure 61 - Peripheral S Register During Input Loading

After loading, the HPS assigns the I, J and K registers the values that correspond to the number of rows in the kernel matrix, number of columns in the input matrix and the number of columns in the kernel matrix respectively. These values are used in the SPP in order to execute its functional algorithm.

Assigning the S register a value of 0x06 will start the matrix multiplication process on the FPGA. The HPS would poll the S register for a value of 0x08 which is the value that the FPGA would assign to the S register upon its completion.

X	X	X	X	X	X	X	X	X	ldout	ldinp	ldker	mfin	mstrt	ldfin	ldstrt
X	X	X	X	X	X	X	X	X	0	0	0	0	1	1	0

Figure 62 - Peripheral S Register During Matrix Multiplication

X	X	X	X	X	X	X	X	X	ldout	ldinp	ldker	mfin	mstrt	ldfin	ldstrt
X	X	X	X	X	X	X	X	X	0	0	0	1	0	0	0

Figure 63 - Peripheral S Register After Matrix Multiplication

Finally, the HPS would assign the S register to the value of 0x48 and begin unloading the output matrix values via the use of I and K for Row and Column indexing. However, a challenge in the fundamentals of HDL multiplication presents itself. The design follows a 16 bit data architecture, however when multiplying two 16 bit numbers the output can be in range of a 32 bit number. Thus, we overcome this challenge by having the internal output register designed as a 32 bit register and when unloading the output the least significant 16 bits would be read from the N register and the most significant 16 bits would be read from the K register. This routine loops until the output matrix is fully unloaded.

X	X	X	X	X	X	X	X	X	ldout	ldinp	ldker	mfin	mstrt	ldfin	ldstrt
X	X	X	X	X	X	X	X	X	1	0	0	1	0	0	0

Figure 64 - Peripheral S Register During Output Unload

When a value of 0x0A is assigned to the S register of the peripheral, the peripheral resets its internal registers as well as issues a reset signal to the SPP component.

X	X	X	X	X	X	X	X	X	ldout	ldinp	ldker	mfin	mstrt	ldfin	ldstrt
X	X	X	X	X	X	X	X	X	0	0	0	1	0	1	0

Figure 65 - Peripheral S Register After Completion/Reset

4.6. System Implementation

Combining all the components mentioned previously into one implementation would allow for a full system implementation. However, this stage is where the system limitations and the challenge of constraints are most prevalent, where precaution and research are rewarded.

The first challenge that presented itself was the idea of fixed point arithmetic. The SPP we designed supported integer multiplication and addition. However, our trained model relied on floating point arithmetic. In order to keep the design feasible, we researched methods of transforming the IEEE 754 floating point standard into a respective fixed point representation. This however proved inefficient due to the sheer amount of parameters that neural networks generate, we would have the challenge of implementing an extremely efficient transformation function. However, after much research, a fundamental method was developed, whereby rounding the input and kernel matrices to the nearest second decimal place and multiplying them by 100 can achieve a transformation of floating point values to their fixed point representations. Whereby the newly generated numbers can be multiplied and added upon in integer form. However, in order to extract the correct result, we divide the output by 10,000. Thus, a rudimentary floating point to fixed point transformation was developed.

Another challenge was one of utilization constraints, the DE10 Standard contains 41.910 Kb of logical elements bandwidth. Shown below is our smallest footprint model's largest matrix sizes required. Even though MobileNet V1 provides a high reduction in parameter size, this is still not sufficient in order to simply load all the values and execute.

Table 8 - Largest Matrix Sizes (Two Dimensional)

Matrix	Largest Kernel Matrix	Largest Input Matrix	Largest Output Matrix
Dimensions	256 x 9 x 16	9 x 4096 x 16	256 x 4096 x 32
Size (Kbit)	36.864	589.824	33554.432

Table 8 shows the kernel and input matrices sporting a relatively small footprint and yet it cannot fit within the DE10's logical elements bandwidth. Moreover, the output matrix would far exceed the board's specifications. Thus, in order to attempt a realistic implementation, the flow of data between the HPS and the FPGA must be throttled. Following is the system implementation where the shared memory is deployed in the form of register components.

Table 9 - Largest Matrix Sizes (One Dimensional)

Matrix	Largest Kernel Matrix	Largest Input Matrix	Largest Output Matrix
Dimensions	1 x 9 x 16	9 x 1 x 16	1 x 1 x 32
Size (Kbit)	0.144	0.144	0.032

Referencing table 8, it can be concluded that the largest shared value of kernel column and input row will not exceed 9. Therefore, in table 9, we designed the registers in such a way that the matrix multiplication is carried in a one dimensional manner. This achieves the required system functionality successfully while utilizing a minimal utilization footprint. However, this approach incurs massive overhead in terms of IO utilization and repeated and inefficient loadings of input

values due to the nature of matrix multiplication. Shown below is the Quartus Utilization Summary for the register based system.

Flow Status	Successful - Thu Jun 03 22:27:50 2021
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	DE10_Standard_GHRD
Top-level Entity Name	DE10_Standard_GHRD
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	3,460 / 41,910 (8 %)
Total registers	5352
Total pins	338 / 499 (68 %)
Total virtual pins	0
Total block memory bits	2,816 / 5,662,720 (< 1 %)
Total DSP Blocks	1 / 112 (< 1 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	1 / 4 (25 %)

Figure 66 - Register Based System Summary

The under utilization of the DE10 board can provide an approach for lower end devices, however in order to fully utilize its performance advancement an iteration on the base design is undertaken, whereby the SPP component relies on block ram memory instead of register based memory that is tightly limited by a small logical element (LE) bandwidth. The BRAM embedded memory (EM) bandwidth is much larger than the LE limit, having approximately 5,662 Kb available. Shown below is the Quartus Utilization Summary for the BRAM based system.

Flow Status	Successful - Thu Jun 03 22:13:08 2021
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	DE10_Standard_GHRD
Top-level Entity Name	DE10_Standard_GHRD
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	3,280 / 41,910 (8 %)
Total registers	4550
Total pins	338 / 499 (68 %)
Total virtual pins	0
Total block memory bits	2,198,656 / 5,662,720 (39 %)
Total DSP Blocks	4 / 112 (4 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	1 / 4 (25 %)

Figure 67 - Block Ram Based System Summary

Contrasting between both utilization summaries discloses the amount of BRAM utilized by the BRAM system and entails an insignificant reduction of 180 bits of logical elements. Having approximately 2195 Kbits consumption of BRAM bandwidth can be unreachable in the case of some low end FPGA devices. The BRAM iteration sports a significant decrease in execution time compared to the register based design. However, referencing table 8, the BRAM iteration would be able to house the kernel and input matrices sufficiently, it would however be unable to house the output matrix due to its sheer relative size. In order to overcome this challenge and with the aid of the dynamic nature of our design, we designed this BRAM iteration to have the HPS process the first four convolutional blocks of our custom CNN MobileNet V1 architecture, and offload the reset of the convolutional blocks to the FPGA. This reduction provided the following largest matrix sizes metrics.

Table 10 - Largest Matrix Sizes (Reduced)

Matrix	Largest Kernel Matrix	Largest Input Matrix	Largest Output Matrix
Dimensions	256 x 9 x 16	9 x 256 x 16	256 x 256 x 32
Size (Kbit)	36.864	36.864	2,097.152

Shown in table 11 is the comparison of the register based, BRAM and the functional simulation implementation which will provide needed contrast on the whole project implementation.

Table 11 - System Implementation Metrics

DE10 Standard	Functional Simulation	Register	BRAM
Accuracy	99.7 %	99.6 %	99.6 %
Total Time (sec)	309.8	1654.3	333.5

The register based implementation is a valid approach, it is however constrained by the inefficient method of memory loading. The BRAM approach more closely resembles the functional simulation in its total time in execution. In the case of the functional simulation, the matrices are already loaded into memory and are processed on command. However, a 23.7 incurred overhead is added to the BRAM approach due to memory loading and unloading from the HPS to FPGA components.

Improvements on the design could find a middle ground between the register approach and the BRAM approach. Utilizing the BRAM based memory for more bandwidth, one could carry out a design as shown in table 12, where the kernel sizes remain as in the register based approach, while the input and output kernels could be adjusted accordingly.

Table 12 - Largest Matrix Sizes (Future Work)

Matrix	Largest Kernel Matrix	Largest Input Matrix	Largest Output Matrix
Dimensions	1 x 9 x 16	9 x 4096 x 16	1 x 4096 x 32
Size (Kbit)	0.144	589.824	131.072

This would introduce a lesser BRAM utilization than the approach taken in table 10, while also having all the convolutional blocks running on the FPGA. Moreover, this would counter the largest disadvantage in the case of the register based approach, whereby the input matrix is repeatedly reloaded. The approach disclosed in table 12 would load the input matrix once and have the kernel rows loaded and multiplied accordingly.

Another method that can introduce better performance would be to treat the HPS as an API. As referenced in table 11, the time taken to complete the functional simulation is relatively long compared to other platforms. Instead of having the HPS both run the model and interface with the FPGA, we could offload the model to a faster platform and communicate with the HPS via sockets. Having the platform that is running the model simply sends the kernel and input matrices to the HPS, and having the HPS interface with the FPGA and return the output back to the device that is running the model. The overhead incurred from the socket transmission and the memory loading/unloading should be studied and accounted for.

5. Aims

This project aims to create a site specific farming design that is power efficient, performant and portable in order to fit the application of plant disease detection in remote agricultural sites. Avoiding mass pesticide use will confirm the intuitive idea that precision agriculture should reduce overall environmental damage, yield more healthier crops and preserve neighboring ecosystems. Not only will the application of inputs such as fertilisers be applied only where and when they are needed, but also the classify the specific type of medicine such as antibacterial or antiviral. This design aims to create an FPGA edge device that is deployed on site which receives images as input and can accurately classify various plant diseases.

6. Design Objectives

Designing and implementing a plant disease detection edge device that is deployed on the FPGA platform that will receive input images and output a classification based on the feature classes discussed previously.

1. Employing a research backed design to fit into applicational constraints.
2. Selecting a CNN architecture whose parameters have a small memory footprint in order to qualify for FPGA synthesis.
3. To run the plant disease detection model on the FPGA we researched how TensorFlow implements their optimized functions and reconstructed them into a custom CNN architecture that supports MobileNet V1.
4. With more freedom in the design, transform 2D convolutions from the spatial domain to the vectorized domain to allow matrix multiplications to represent 2D convolutions.
5. Create a custom single purpose processor to allow for accelerated matrix multiplications on DSP chips and synthesize it on the FPGA board.
6. Create a custom Peripheral slave for the FPGA fabric and the ARM CPU located on the board.
7. Interface the ARM CPU and the FPGA fabric and reach a system implementation.
8. Evaluate and iterate.

7. Design Milestones

In order to achieve our designs and implementations to meet the requirements of the project application, the following specifically planned milestones need to be checked off.

1. Fundamentals: Encompass the principals of Convolutional Neural Network (CNN).
2. Review: Gather data regarding current advancements and ongoing research on image feature detection and applications in plant leaf disease detection.
3. Tools: Make an educated decision on the framework and tools to use to create a model of the inference machine.
4. Implementation: Apply model to FPGA device whilst understanding memory and other limitations will steer the implementation.
5. Evaluation: Iteration and performance evaluation.

8. Design Options

Research was initiated in order to discover the top of the line deployments of CNNs on FPGAs, the following options were researched. Due to constraints in the design, some of the approaches listed below are not included in the report. The applicable approaches were compared and analysed in order to make an educated decision on which option to comply with in order to achieve our final design.

1. Embedded Devices
    a. RaspberryPi
    b. Jetson Nano
2. GPUs
3. FPGA
    a. HLS
        i. Vivado
        ii. OpenCL
        iii. Raw
        iv. OpenVino
        v. Nios
    b. RTL

9. Assumptions and Design Codes

In this section we expose our implementations' repositories.

9.1. MNIST VHDL Implementation Repository

Following is the link for the code to the implementation outlined in section 3.

https://drive.google.com/file/d/1p-PIcrqtl4hWUZ9iOt7v-oAm_t3MC3-9/view?usp=sharing

9.2. Applicational Implementation Repository

Following is the link for the repository to the implementation outlined in section 4.

https://drive.google.com/drive/folders/1TcJyFZ0jjb1SPf-FspdV3fkci-oT0lkh?usp=sharing

10. References

[1] A. L. P. d. Ocampo and E. P. Dadios, "Mobile Platform Implementation of Lightweight Neural Network Model for Plant Disease Detection and Recognition," *2018 IEEE 10th International Conference on Humanoid, Nanotechnology, Information Technology,Communication and Control, Environment and Management (HNICEM)*, Baguio City, Philippines, 2018, pp. 1-4, doi: 10.1109/HNICEM.2018.8666365.

[2] M. Francis and C. Deisy, "Disease Detection and Classification in Agricultural Plants Using Convolutional Neural Networks — A Visual Understanding," *2019 6th International Conference on Signal Processing and Integrated Networks (SPIN)*, Noida, India, 2019, pp. 1063-1068, doi: 10.1109/SPIN.2019.8711701.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. “ImageNet classification with deep convolutional neural networks”. *In Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12)*. Curran Associates Inc., Red Hook, NY, USA, 1097–1105.

[4] M. F. Uddin, "Addressing Accuracy Paradox Using Enhanced Weighted Performance Metric in Machine Learning," *2019 Sixth HCT Information Technology Trends (ITT)*, Ras Al Khaimah, United Arab Emirates, 2019, pp. 319-324, doi: 10.1109/ITT48889.2019.9075071.

[5] A. Mikołajczyk and M. Grochowski, "Data augmentation for improving deep learning in image classification problem," 2018 International Interdisciplinary PhD Workshop (IIPhDW), Świnoujście, Poland, 2018, pp. 117-122, doi: 10.1109/IIPHDW.2018.8388338.

[6] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (*FPGA '16*). Association for Computing Machinery, New York, NY, USA, 16–25. DOI:https://doi.org/10.1145/2847263.2847276

[7] Yongmei Zhou and Jingfei Jiang, "An FPGA-based accelerator implementation for deep convolutional neural networks," 2015 4th International Conference on Computer Science and Network Technology (ICCSNT), Harbin, China, 2015, pp. 829-832, doi: 10.1109/ICCSNT.2015.7490869.

[8] M. K. Hamdan and D. T. Rover, "VHDL generator for a high performance convolutional neural network FPGA-based accelerator," *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig),* Cancun, Mexico, 2017, pp. 1-6, doi: 10.1109/RECONFIG.2017.8279827.

[9] Kyriakos, A. (2019). High Performance Accelerator for CNN Applications. *2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)* (pp. 135-140). Athens: IEEE.

[10] D. Steinkraus, I. Buck, and P. Y. Simard, “Using GPUs for machine learning algorithms,” in Eighth International Conference on Document Analysis and Recognition (ICDAR 05), 2005.

[11] K. Ovtcharov, O. Ruwase, J. Kim, J. Fowers, K. Strauss, and E. S. Chung, “Accelerating Deep Convolutional Neural Networks Using Specialized Hardware,” Microsoft Res. Whitepaper, 2015.

[12] Carballo-Hernández, W., Pelcat, M. and Berry, F., 2021. *Why is FPGA-GPU Heterogeneity the Best Option for Embedded Deep Neural Networks?*. Retrieved from: https://arxiv.org/abs/2102.01343.

[13] Wei Dai, Daniel Berleant. Benchmarking Contemporary Deep Learning Hardware and Frameworks: A Survey of Qualitative Metrics. 2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI), Dec 2019, Los Angeles, United States. pp.148-155, ff10.1109/CogMI48466.2019.00029ff. ffhal-02501736.

[14] Andrew Ng, Kian Katanforoosh and Younes Bensouda Mourri. (2020). Deep Learning Specialization. *Deep Learning AI Courses.* Retrieved from https://www.coursera.org/specializations/deep-learning

[15] Howard, A., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *ArXiv, abs/1704.04861*.

[16] Bai, K. (2019). A Comprehensive Introduction to Different Types of Convolutions in Deep Learning. *Towards Data Science*. Retrieved from https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215.

[17] Jin, Y. A comprehensive survey of fitness approximation in evolutionary computation. *Soft Computing* 9, 3–12 (2005). https://doi.org/10.1007/s00500-003-0328-5

[18] TensorFlow OpenSource Contributors. Keras Applications. *TensorFlow's MobileNet V1 Architecture Implementation*. Retrieved from https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/keras/applications/mobilenet.py

[19] Stuart J. Russell, Peter Norvig (2010) *Artificial Intelligence: A Modern Approach, Third Edition*, Prentice Hall ISBN 9780136042594.

[20] Xilinx XST user guide. (2009). Xilinx, Inc.

11. Summary

Smart Automation Controller for Precision Agriculture

Design Project

Tarek Omar - 153143

Laila Hatem - 152493

Abdalla Mohamed - 164630

Ahmed Essam Rady - 129625

Abdelrahman Mohammad - 157900

Dr. Ihab Adly - Supervisor

Faculty of Engineering

Electrical Engineering Department

(Computer Engineering Programme)

11.1. Abstract

The significant rise in the deep neural networks in image detection and classification applications created an emergent industry that is led by applicational innovation. This project focuses on off the grid agricultural farms, where the need to detect plant diseases through plant medical centers is limited. In the review section, this report covers the fundamental building blocks that lead to the creation of CNN architectures. It also covers how Field Programmable Gate Arrays (FPGAs) are the platform of choice for efficient and responsive systems that edge devices require. Moreover, a fully VHDL implementation of the MNIST dataset was developed. In the MNIST VHDL Implementation section, we discuss the major layers and how they were conjoined to form an overall system. By using functional, testbench and timing simulations, we can pinpoint and understand the low level implications of creating such a system in base the RTL flow. Furthermore, the Applicational Implementation section will discuss our methodology in order to meet the requirements of the application, the many challenges that were met and the unique approaches generated to overcome them.

11.2. Problem Statement

Avoiding mass pesticide use will confirm the intuitive idea that precision agriculture should reduce overall environmental damage, yield more healthier crops and preserve neighboring ecosystems. Not only will the application of inputs such as fertilisers be applied only where and when they are needed, but also the classify the specific type of medicine such as antibacterial or antiviral. This design aims to create an FPGA edge device that is deployed on site which receives images as input and can accurately classify various plant diseases.

11.3. Objectives

Designing and implementing a plant disease detection edge device that is deployed on the FPGA platform that will receive input images and output a classification.

1. Employing a research backed design to fit into applicational constraints.
2. Selecting a CNN architecture whose parameters have a small memory footprint in order to qualify for FPGA synthesis.
3. To run the plant disease detection model on the FPGA, we researched how TensorFlow implements their optimized functions and reconstructed them into a custom CNN architecture that supports MobileNet V1.
4. With more freedom in the design, we transformed 2D convolutions from the spatial domain to the vectorized domain to allow matrix multiplications to represent 2D convolutions.
5. Create a custom single purpose processor to allow for accelerated matrix multiplications on DSP chips and synthesize it on the FPGA board.
6. Create a custom Peripheral slave for the FPGA fabric and the ARM CPU located on the board.
7. Interface the ARM CPU and the FPGA fabric and reach a system implementation.
8. Evaluate and iterate.

11.4. Review

Understanding the fundamentals of CNNs allows for low level understanding of the underlying operations that occur.

Input layers of CNNs are concerned with acquiring the pixel values of the input image; in three dimensions (RGB). Input tensor is calculated via: (#images in dataset) x (image dimensions). Where image pixel values populate the tensor.

The convolution operation is executed by performing dot products, as shown in the below figure. The kernel tensor dimensions are calculated via:
(kernel dimensions) x (image depth) x (#kernels) with the weights populating.

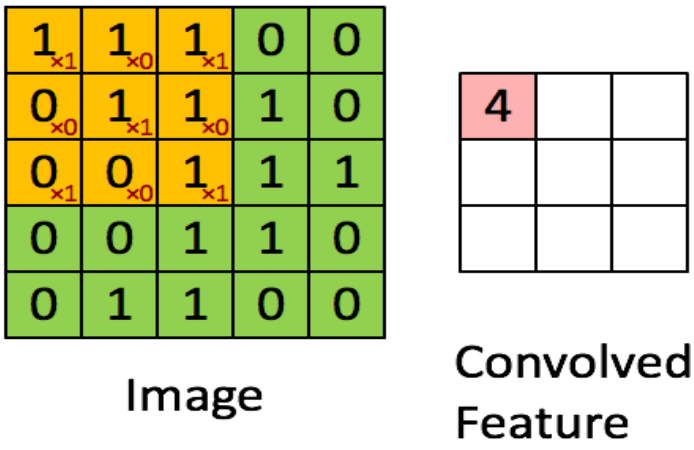


Figure 2 - Convolution Operation (Green: Input, Yellow: Kernel, Red: Output) [14]

The convolution operation outputs requires dimensions of size (#images) x (n_H) x (n_W) x (#kernels) where n_H and n_W are preprocessed values obtained by using the equation shown below.

$$n_H = \lfloor \frac{n_{H_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$
$$n_W = \lfloor \frac{n_{W_{prev}} - f + 2 \times pad}{stride} \rfloor + 1$$
$$n_C = \text{number of filters used in the convolution}$$

Figure 3 - Preprocessing of Convolution Output Dimensions [14]

ReLU introduces nonlinearity in order to create complex signals, in order to create complex functions that can correctly classify between features to appropriately infer between them, shown in figure 5.

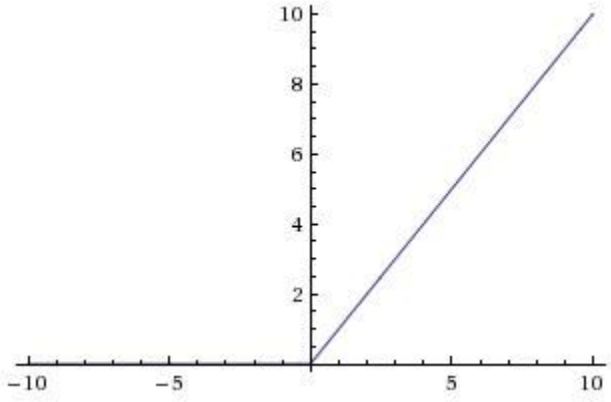
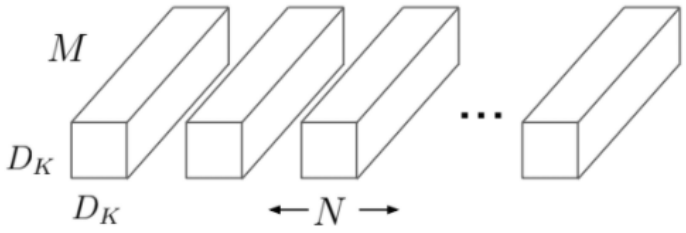


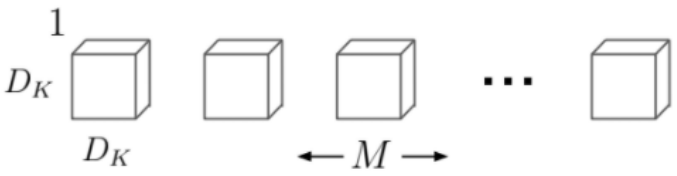
Figure 5 - ReLU function Graph

FPGAs showed 30% power decrement and 4% to 26% subtraction in latency, whereas in ShuffleNetv2 they uncovered a power decrement of 25% and a latency decrement of 21%, and SqueezeNet (21%-28% energy subtraction, same latency) [12].

Mobilenet models are researched in order to outline their advantages and drawbacks as compared to standard convolutional models.



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters

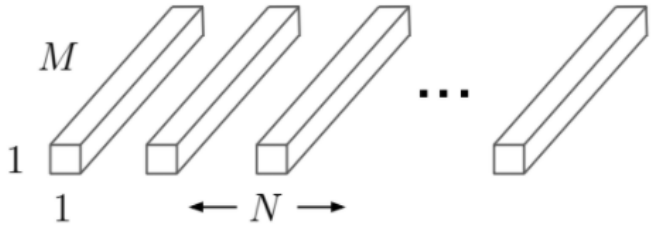


Figure 9 - Output Depth of Standard vs Depthwise Convolutions [15]

Complexity of Standard convolution operation:

$$D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F$$

Complexity of Depthwise convolution:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F$$

Complexity of Depthwise Convolution + Pointwise convolution:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F$$

Depthwise separable convolution over standard convolution complexity reduction:

$$\frac{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F}$$

$$= \frac{1}{N} + \frac{1}{D_K^2}$$

Thus, as shown, depthwise separable convolution is objectively mathematically less complex than standard convolution.

Following is a general attribute table for the researched CNN implementations on FPGA.

Table 4 - HLS General Table

Metric	OpenCL	Raw SoC	Vivado	RTL
Host lang	OpenCL C	C/Python	C/C++	VHDL
Accel. lang	Compute Kernel	VHDL	C/C++	VHDL
Host interface	SDK/DMA	Avalon/AXI/ DMA	Avalon/AXI/ DMA	DMA/Not required depending on model size

Following is a utilization table, where different implementations are aggregated.

Table 5 - HLS Utilization Table

Metric	OpenCL [6]	Vivado [7]	RTL [8]
Device	DE5-Net	xczu9eg-ffvb1156-2-i	Virtex VC709
ALUT/ALMs	49%	11%	66%
FF	-	6%	31%
BRAM	74%	5%	68%
DSP	100%	5%	57%

Following is a performance table

Table 6 - HLS Performance Table

Metric	OpenCL [6]	Vivado [7]	RTL [8]
Device	DE5-Net	xczu9eg-ffvb1156-2-i	Virtex VC709
Precision	8 bit	11 bit	16 bit
# Op. per Image	1.46 GOP	410,758 Cycles	611.52 GOP
Inference Time (Approx.)	45.7 ms	3.58 ms	2.41 ms

11.5. Methodology

This project takes on two major methodologies, the first is to create an MNIST model that is developed using VHDL. The second approach is an implementation of the Raw approach that is developed to solve the applicational problem of the project.

The MNIST implementation's main aim is to design a convolutional neural network by using HDL. The implemented network mainly consists of four layers, the input layer, the convolutional layer, the pooling layer and the fully connected layer. The input layer consists of a counter and two memory ROMs that are initialized by external files that hold the values of 6x6 grey scale input image's pixels and the values of 3x3 input kernel. The convolution layer mainly used to perform the convolution operation between all the 3x3 convolution windows resulted from the input layer with the input kernel. Moving to the pooling layer, its main task is to select the maximum value of 2x2 windows with stride of two. Finally the fully connected layer is used to perform three mathematical operations in order to output the final softmax layer that represents the output prediction of the neural network.

In order to achieve the applicational implementation, we begin by selecting CNN architectures that maintain a high accuracy threshold while also leaving a small storage footprint. After selecting appropriate architectures, training the model introduced many challenges that are documented in turn. Due to the dynamic nature of this project, we had to develop a low level custom CNN architecture that supports MobileNet V1 on Python, without the aid of TensorFlow libraries. This was accomplished by reviewing how TensorFlow implements their optimized functionalities and reconstructing them, with our approaches documented in turn. A full system functional simulation was developed in Python and through the use of pointer arrays and threading libraries we were able to emulate a shared memory component and simulate realistic real time components running and interfacing concurrently. By following industry techniques, we developed a custom single purpose processor for applying a matrix multiplication algorithm to VHDL. Given its inputs and a start signal, the RTL driven algorithm would run synthesized on the FPGA. Exploring an interfacing medium, we researched Avalon Memory Mapped Interfaces, this allowed for a straightforward memory mapped peripheral design. Furthermore, to utilize the memory mapped medium we developed a custom peripheral component that wraps the single purpose processor and exposes the peripheral (Slave) to the interface medium for communication of the HPS and FPGA (Masters). Moreover, we developed a datasheet for the peripheral that documents the usability of the component. And lastly, joining all the designed components into the system implementation, this introduced the applications' limitations into clear view. We overcome the limitations and apply a base implementation, an iteration upon it and discuss their limitations and compare their results, we also introduce future work iterations that could provide a balance point between our two iterations.

11.6. Approaches

To allow for a modular approach, a custom model training, validation and evaluation suite was implemented on Google cloud servers to allow for parallel, quick and reliable model creation.

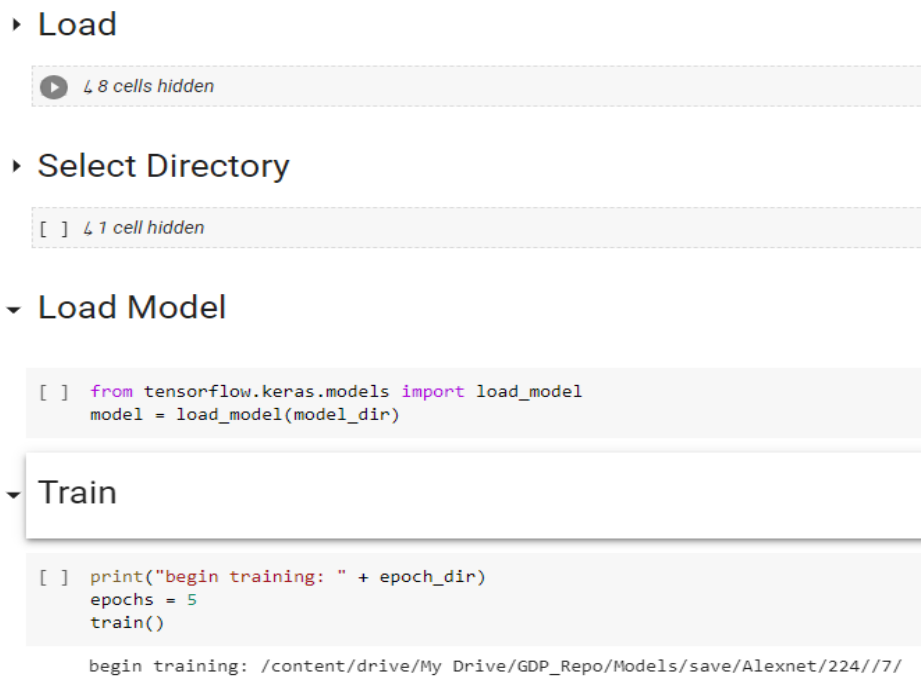


Figure 38 - Model Training Suite

According to the theory of model fitness approximation, [17] argues for an empirical approach of per model basis for selecting the most appropriate epoch that approximates the whole model fitness.

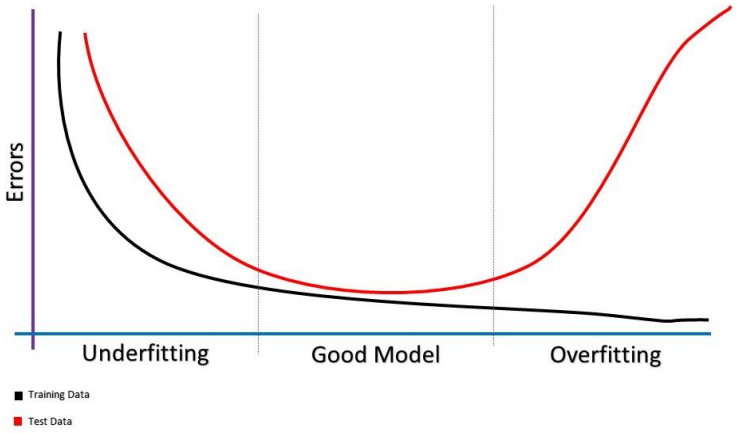


Figure 41 - Model Fitness Reference Graph

In order to verify an accepted state, for each epoch we recorded the accuracy achieved and the loss incurred. Shown in Figure 39 & 40, are the accuracy and loss graphs for our smallest storage and processing footprint model of MobileNet V1 with 128 resolution and 0.25 width multiplier.

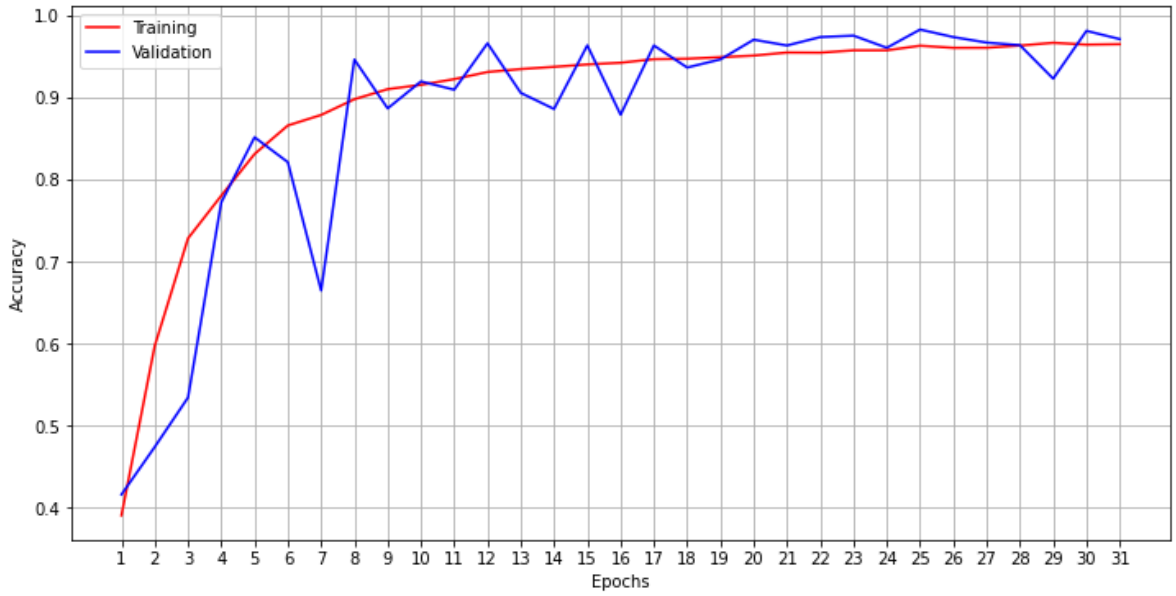


Figure 39 - Epoch/Accuracy Graph

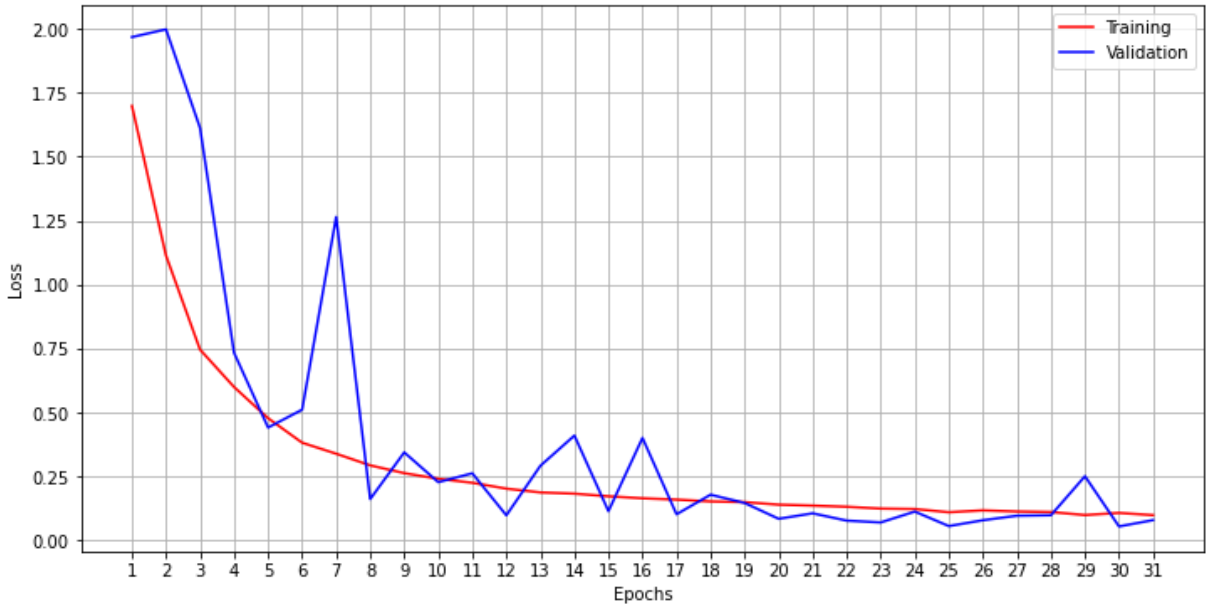


Figure 40 - Epoch/Loss Graph

The dataset used is from PlantVillage, where our models were trained on 22,680 images, which were approximately equally split between the 12 classes to avoid accuracy paradox. [4] Following are the 12 classes that our models can classify and infer:

Table 1 - Plant Disease Model Classes

Apple	**Corn**	**Tomato**
Healthy	Healthy	Healthy
Scab	Cercospora leaf spot	Mosaic virus
Rust	Rust	Yellow leaf curl virus
Black rot	Blight	Early blight

Comparing our implementation to other attempts to tackle similar applications provides feasibility.

Table 3 - Model Comparisons

Model	Mobilenet		Mobilenet [1]	Custom [2]
Top 1 Accuracy	98%	98%	89%	88.7%
Dataset Size	22,680	22,680	6,970	3,663
#Classes	12	12	6	Binary
Input size(Width multiplier)	224x224(0.25)	128x128(0.25)	224x224(0.5)	64x64
#Parameters	221,628	221,628	1.3 million	45,281
Parameters' size	1MB	1MB	5.46 MB	45K
Cloud infer time	0.798	0.754		
Cloud total time	6.784	6.491		
RPI infer time	0.840	0.266	Mobile 0.406	GPU Tesla -
RPI total time	1.051	0.462		

It can be concluded that compared to the current edge, our memory and storage footprint models find balance between accuracy and efficiency.

TensorFlow is elegant and useful in many applications and for debugging, however it abstracts alot of its functionalities for the sake of simplicity. Thus a custom approach that aims to understand the optimized implementations of the TensorFlow functions and reconstruct them without the use of any of the TensorFlow libraries was undertaken.

In order to begin tackling such a problem, we first devised a list of the building blocks that are used to create the MobileNet V1 architecture [15], shown in the below figure.

1.	Conv2D	tf.nn.conv2d | TensorFlow Core v2.4.1
2.	Depthwise Conv2D	tf.nn.depthwise_conv2d | TensorFlow Core v2.4.1
3.	BatchNorm	tf.nn.batch_normalization | TensorFlow Core v2.4.1
4.	ReLU	tf.nn.relu | TensorFlow Core v2.4.1
5.	ZeroPadding2D	tf.pad | TensorFlow Core v2.4.1
6.	GlobalAveragePooling2D	tf.nn.avg_pool2d | TensorFlow Core v2.4.1
7.	Reshape	tf.reshape | TensorFlow Core v2.4.1
8.	SoftMax	tf.nn.softmax | TensorFlow Core v2.4.1

Figure 44 - Building Block Documentation

Shown is the TensorFlow documentation for their Conv2D implementation.

Given an input tensor of shape `batch_shape + [in_height, in_width, in_channels]` and a filter / kernel tensor of shape `[filter_height, filter_width, in_channels, out_channels]`, this op performs the following:

1. Flattens the filter to a 2-D matrix with shape `[filter_height * filter_width * in_channels, output_channels]`.
2. Extracts image patches from the input tensor to form a *virtual* tensor of shape `[batch, out_height, out_width, filter_height * filter_width * in_channels]`.
3. For each patch, right-multiplies the filter matrix and the image patch vector.

Figure 44 - TensorFlow Conv2D Implementation Documentation

From this description, we have the knowledge that TensorFlow leverages the use of vectorization via the use of patch extraction to achieve their Conv2D implementation.

Given a 3 channel image with dimensions 5x4, shown in figure 45. Highlighted in green is an example of a 3x3 kernel situated at the start of the input image.

```
[0,1,2,3]
[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]
[36,37,38,39]

[40,41,42,43]
[44,45,46,47]
[48,49,50,51]
[52,53,54,55]
[56,57,58,59]
```

Figure 45 - Image Example

Shown in figure 46 is the output of our custom implementation of the vectorization function.

```
inp mat
[[[ 0.  1.  4.  5.  8.  9.]
  [ 1.  2.  5.  6.  9. 10.]
  [ 2.  3.  6.  7. 10. 11.]
  [ 4.  5.  8.  9. 12. 13.]
  [ 5.  6.  9. 10. 13. 14.]
  [ 6.  7. 10. 11. 14. 15.]
  [ 8.  9. 12. 13. 16. 17.]
  [ 9. 10. 13. 14. 17. 18.]
  [10. 11. 14. 15. 18. 19.]]

 [[20. 21. 24. 25. 28. 29.]
  [21. 22. 25. 26. 29. 30.]
  [22. 23. 26. 27. 30. 31.]
  [24. 25. 28. 29. 32. 33.]
  [25. 26. 29. 30. 33. 34.]
  [26. 27. 30. 31. 34. 35.]
  [28. 29. 32. 33. 36. 37.]
  [29. 30. 33. 34. 37. 38.]
  [30. 31. 34. 35. 38. 39.]]

 [[40. 41. 44. 45. 48. 49.]
  [41. 42. 45. 46. 49. 50.]
  [42. 43. 46. 47. 50. 51.]
  [44. 45. 48. 49. 52. 53.]
  [45. 46. 49. 50. 53. 54.]
  [46. 47. 50. 51. 54. 55.]
  [48. 49. 52. 53. 56. 57.]
  [49. 50. 53. 54. 57. 58.]
  [50. 51. 54. 55. 58. 59.]]]
(3, 9, 6)
```

Figure 46 - Vectorized Image Example

As can be deduced, the vectorized representation of the input image carries increased storage footprint, yet leverages increased performance. Shown in table 4, a comparison between the standard spatial convolution, our vectorized implementation and the TensorFlow implementation.

Table 4 - Conv2D Implementation Metrics

Metrics	Spatial	Vectorized	TF
Exec time (126x126x32)	4.9	0.8	0.28

Shown in table 5 are the functions housed inside the LL Functions file.

Table 5 - Low Level Functions

Input Reshape	Kernel Reshape	Pad	Batch Norm	ReLU	SoftMax	ZeroPad	Load Weights	Load File	Display

Shown in table 6 are the High Level Functions and the Low Level Functions that they utilize.

Table 6 - High Level Functions

Conv Block	Conv2D	Conv2D Parallel Reshape
Load Weights, Conv2D, Batch Norm, ReLU, ZeroPad	Pad, Conv2D Parallel Reshape, Numpy.MatMul	Input Reshape, Kernel Reshape

In order to execute component testing, we implemented the algorithm that is being utilized to design our custom Single Purpose Processor in Python, shown in figure 47.

```python
# VHDL code emulation
def mat_mult(temp_ker_mat, temp_inp_mat, ker_row, ker_col, inp_row, inp_col):

  output = np.zeros([ker_row, inp_col])

  for i in range(ker_row):
    for j in range(inp_col):
      for x in range(ker_col):
        output[i][j] += (temp_ker_mat[i][x] * temp_inp_mat[x][j])

  return output
```

Figure 47 - Matrix Multiplication Algorithm

The mat_mult function shown in figure 47, very closely emulates our single purpose processor.

Presented in the following figures are the block diagrams that aided the implementation of our system functional simulation.

Shown in figure 48 is the interfacing between the UI thread and the socket reference.

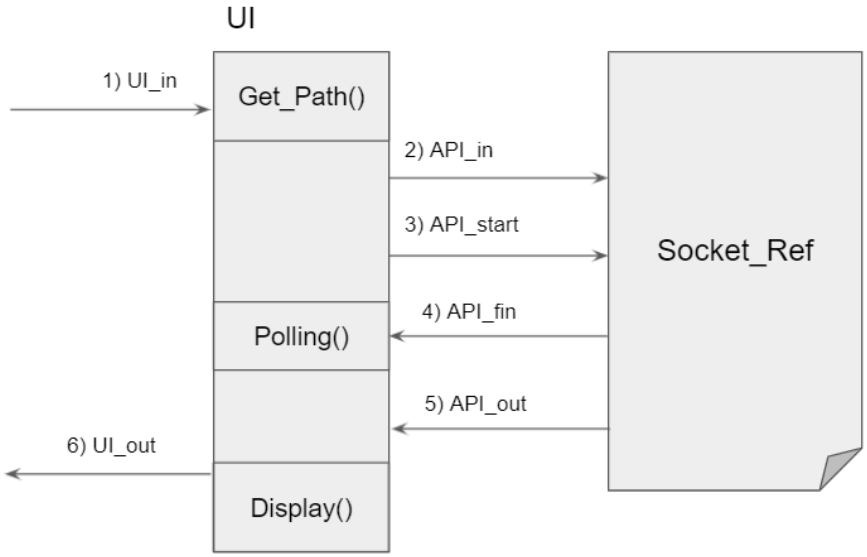


Figure 48 - UI & Socket interfacing

Figure 49 entails the interaction between the socket reference and the API.

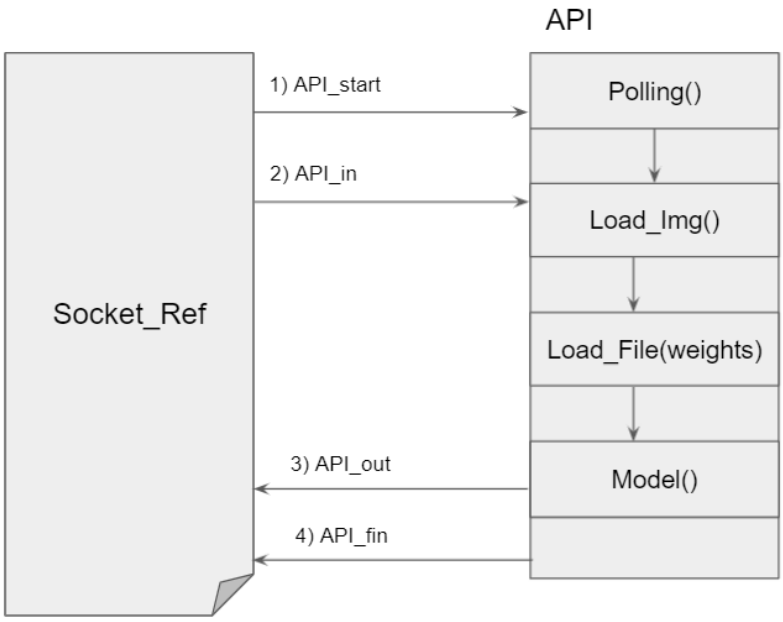


Figure 49 - Socket & API interfacing

Figure 50 discloses the functionality of the Model Function.

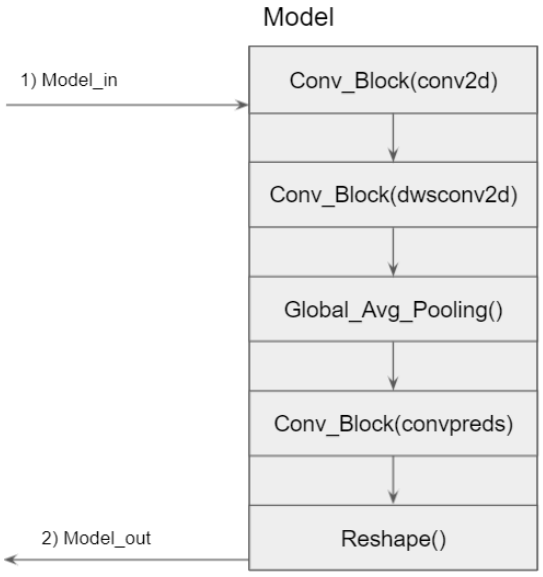


Figure 50 - MobileNet V1 Model

Figure 51 shows the inner workings of the Conv Block function, which is introduced in the MobileNet V1 whitepaper .

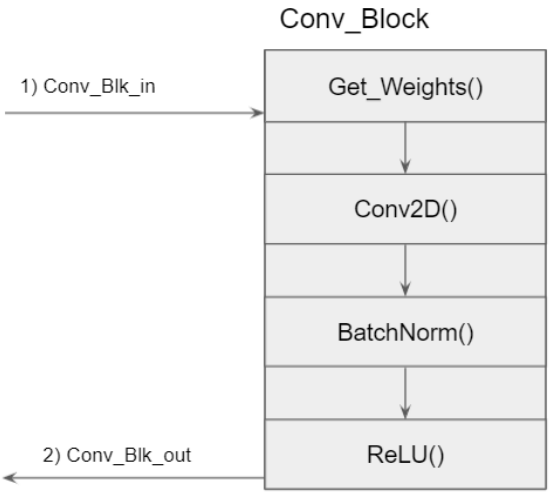


Figure 51 - Convolutional Block Function

Through understanding of the low level functionality of both types of convolutions, we were able to infer that there are only two differing factors that can be accounted for between Conv2D and DW Conv2D. These two factors can be accounted for at the end of our Conv2D implementation. Figure 52 shows the Conv2D function.

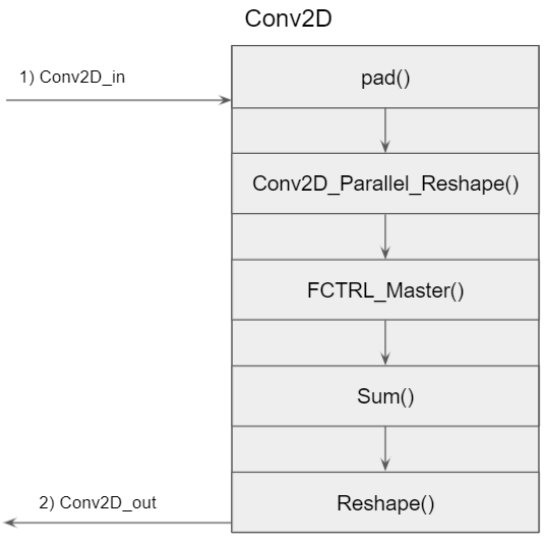


Figure 52 - Unified Convolutional Function

Finally, by functionally simulating the FCTRL components we can realistically design a protocol for proper API to FPGA interfacing. Starting with the flow control master function shown in figure 54.

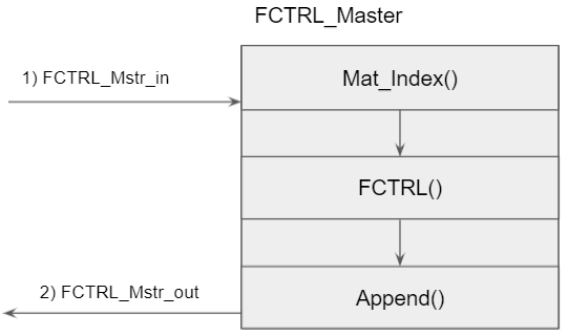


Figure 54 - Flow Control Master

The way the FCTRL function interacts with the shared memory component is simple, it assigns the input values to the shared memory and asserts FPGA_start then polls for FPGA_fin and on assertion returns the FPGA_out value. As shown in figure 55.

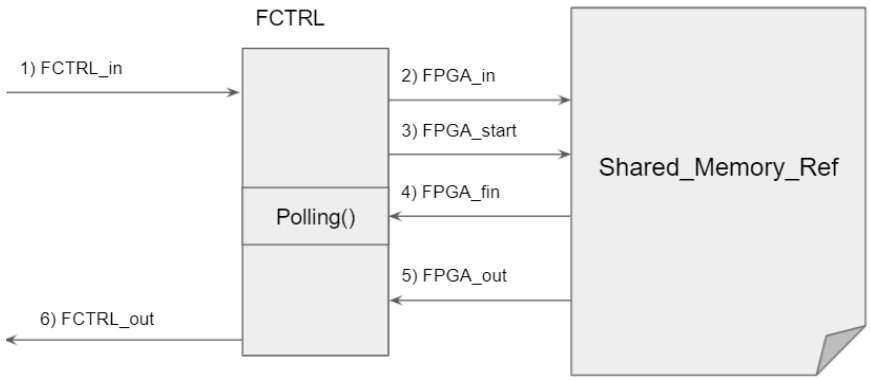


Figure 55 - Flow Control & Shared Memory Interfacing

On the other side, as shown in figure 56, the FPGA is polling the FPGA_start and on assertion, takes the values in FPGA_in and begins its Matrix Multiplication routine. On Completion, the matrix multiplication output is assigned as well as the FPGA_fin asserted.

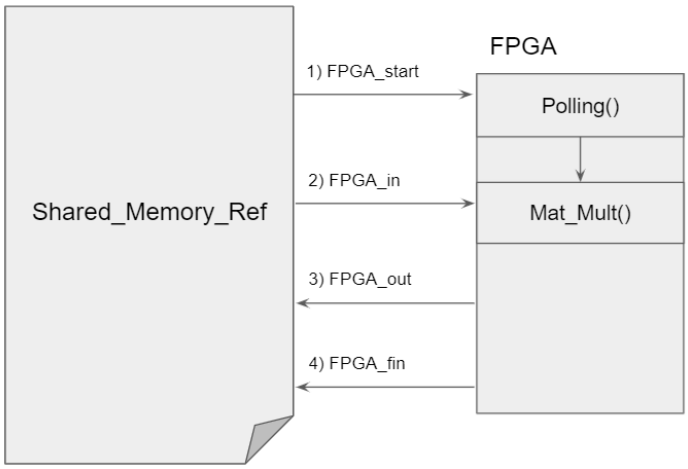


Figure 56 - Shared Memory & FPGA Interfacing

By following the thread of execution, one can understand the full functionality of our designed system. This design was used to implement the system functional simulation. Following are the aggregate results of the functional simulation running on Intel(R) Core(TM) i7-8700, Google Cloud and the Arm Cortex A9 of the DE10 Standard FPGA.

Table 7 - Functional Simulation Metrics

Device	Intel i7-8700	Google Cloud	Arm Cortex A9
Accuracy	99.7 %	99.7 %	99.7 %
Total Time (sec)	13.7	27.4	309.8

In order to create the single purpose processor that attempts matrix multiplication we implemented algorithms matrix multiplications. This algorithm multiplies A and B and stores the result in C. A and B have dimensions of m1 x m2 and n1 x n2, respectively.

Input parameters: A, B

Output parameters: C

```
Matrix_product_rectangular(A, B, C, m1, m2, n1, n2){

    for i = 0 to m1 - 1
        for j = 0 to n2 - 1 {
            C[i][j] = 0
            for k = 0 to m2 - 1
                C[i][j] = C[i][j] + A[i][k] * B[k][j]
        }
}
```

Time complexity: $O(n^3)$.

Auxiliary space: O (m1 * n2)

This algorithm enables the design of a digital circuit that applies rectangular matrix multiplication in RTL. The RTL design will be in FSMD (data path + FSM) architecture.

This figure shows the architecture of the data path.

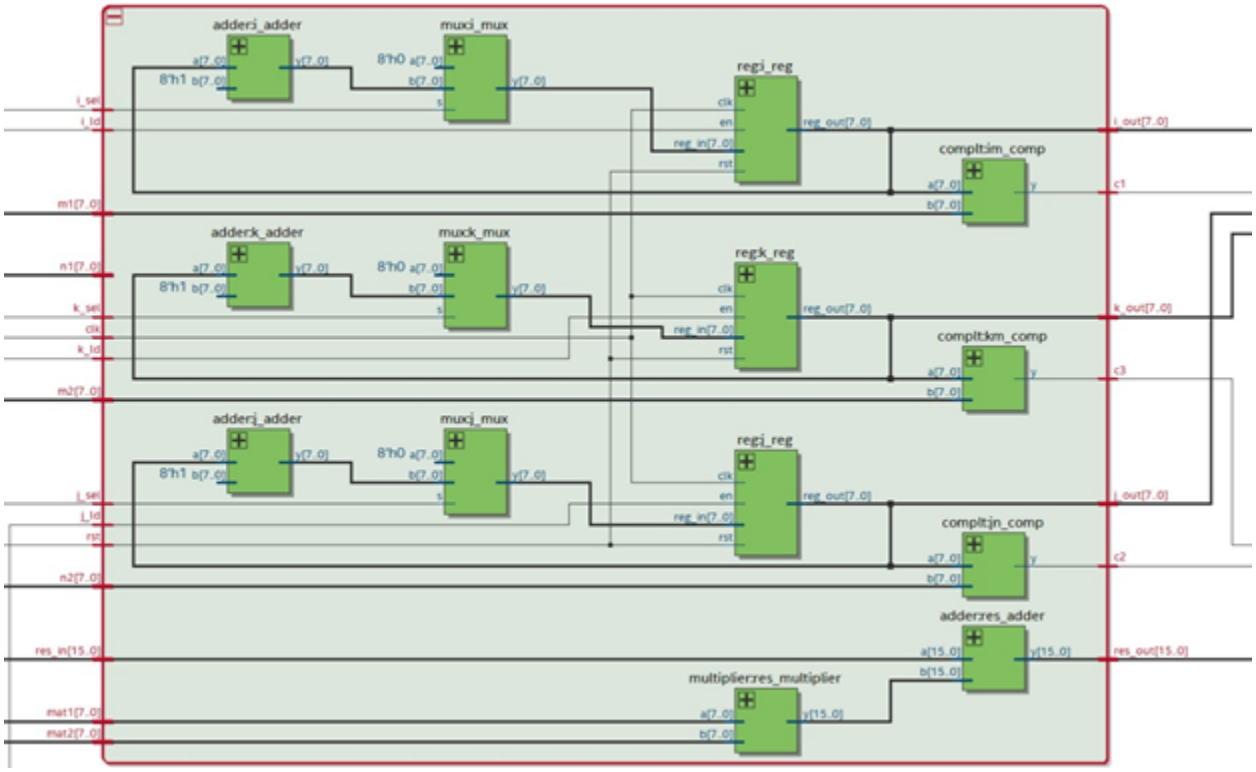


Figure [1].

This figure shows the architecture of the FSM.

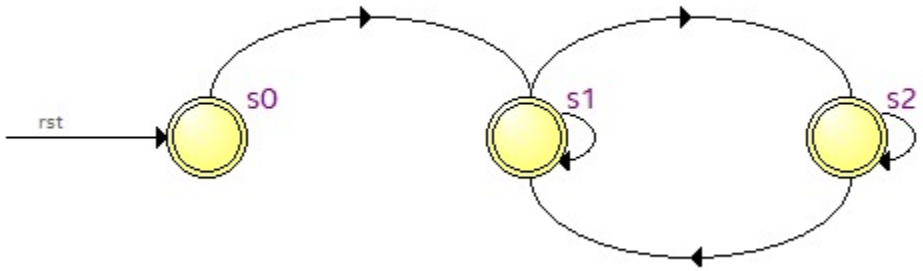


Figure [2].

	Source State	Destination State	Condition
1	s0	s1	
2	s1	s2	(start)
3	s1	s1	(!start)
4	s2	s2	(c1)
5	s2	s1	(!c1)

Figure [3].

This table shows the transitions from one state to another.

The truth table is implemented given the conditions of the loops.

CONTROL SIGNALS TRUTH TABLE

C1	C2	C3	wr	k_ld	k_sel	i_ld	i_sel	J_ld	J_sel
0	0	0	0	0	X	0	X	0	X
0	0	1	0	0	X	0	X	0	X
0	1	0	0	0	X	0	X	0	X
0	1	1	0	0	X	0	X	0	X
1	0	0	0	1	0	1	1	1	0
1	0	1	0	1	0	1	1	1	0
1	1	0	0	1	0	0	X	1	1
1	1	1	1	1	1	0	X	0	X

Figure [4].

The top level of the architecture should be like in this figure.

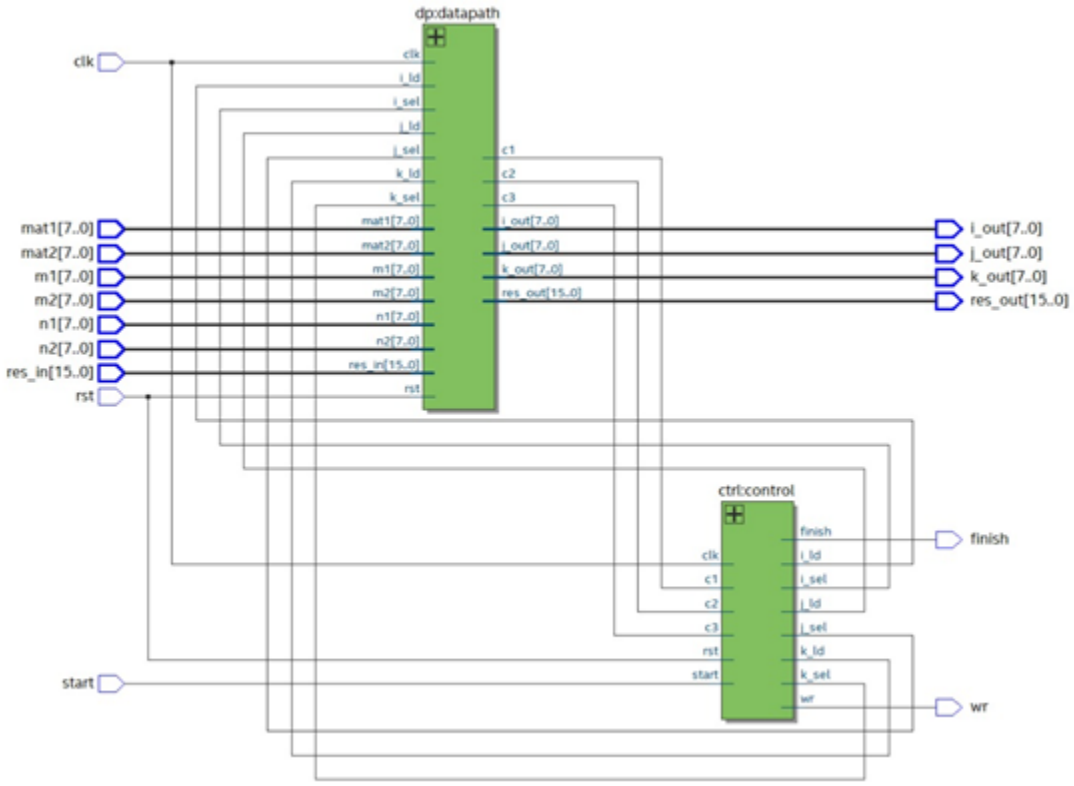


Figure [5].

As it is shown in the figure, the data path outputs the c1, c2, and c3 signals to the controller and takes i_ld, j_ld, k_ld, i_sel, j_sel, and k_sel from the controller. To implement this architecture on given matrices, the two matrices should be stored in ROMs / rams.

With the knowledge of peripheral terminology; we want to create a slave peripheral, whereby it has two masters; the HPS and the FPGA. Following is the block diagram for the custom peripheral.

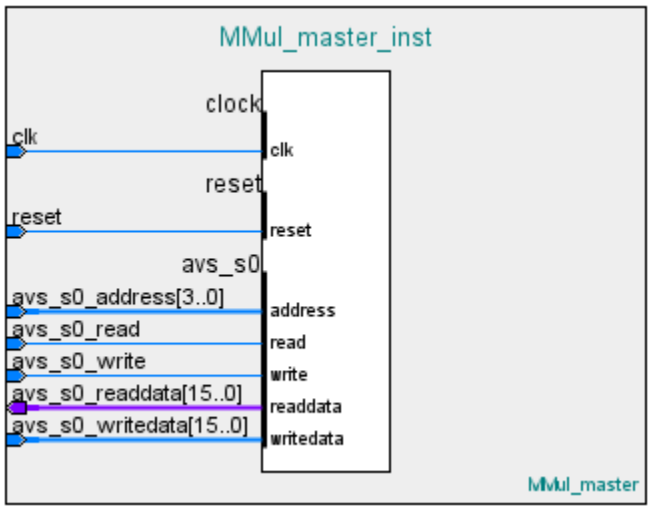


Figure 57 - Peripheral Block Diagram

Our design follows a 16 bit data word register width to allow for reduced processing footprint and minimal bus width. Following are the peripheral registers.

I	J	K	N	S

Figure 58 - Peripheral Registers

Setting the S register to the corresponding hex value will utilize the I, J, K and N registers in different manners.

7) Loading Kernel Matrix (0x11)
8) Loading Input Matrix (0x21)
9) Matrix Multiplication Start (0x06)
10) Matrix Multiplication Finish (0x08)
11) Unloading Output Matrix (0x48)
12) Completion/Reset (0x0A)

Shown below is the structure of the S register.

X	X	X	X	X	X	X	X	X	ldout	ldinp	ldker	mfin	mstrt	ldfin	ldstrt

Figure 59 - Peripheral S Register Structure

11.7. Results

Combining all the components mentioned previously into one implementation would allow for a full system implementation. However, this stage is where the system limitations and the challenge of constraints are most prevalent, where precaution and research are rewarded.

Even though MobileNet V1 provides a high reduction in parameter size, this is still not sufficient in order to simply load all the values and execute.

Table 8 - Largest Matrix Sizes (Two Dimensional)

Matrix	Largest Kernel Matrix	Largest Input Matrix	Largest Output Matrix
Dimensions	256 x 9 x 16	9 x 4096 x 16	256 x 4096 x 32
Size (Kbit)	36.864	589.824	33554.432

Table 8 shows the kernel and input matrices sporting a relatively small footprint and yet it cannot fit within the DE10's logical elements bandwidth. Moreover, the output matrix would far exceed the board's specifications. Thus, in order to attempt a realistic implementation, the flow of data between the HPS and the FPGA must be throttled. Following is the system implementation where the shared memory is deployed in the form of register components.

Table 9 - Largest Matrix Sizes (One Dimensional)

Matrix	Largest Kernel Matrix	Largest Input Matrix	Largest Output Matrix
Dimensions	1 x 9 x 16	9 x 1 x 16	1 x 1 x 32
Size (Kbit)	0.144	0.144	0.032

Referencing table 8, it can be concluded that the largest shared value of kernel column and input row will not exceed 9. Therefore, in table 9, we designed the registers in such a way that the matrix multiplication is carried in a one dimensional manner. This achieves the required system functionality successfully while utilizing a minimal utilization footprint. However, this approach incurs massive overhead in terms of IO utilization and repeated and inefficient loadings of input values due to the nature of matrix multiplication. Shown below is the Quartus Utilization Summary for the register based system.

Flow Status	Successful - Thu Jun 03 22:27:50 2021
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	DE10_Standard_GHRD
Top-level Entity Name	DE10_Standard_GHRD
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	3,460 / 41,910 (8 %)
Total registers	5352
Total pins	338 / 499 (68 %)
Total virtual pins	0
Total block memory bits	2,816 / 5,662,720 (< 1 %)
Total DSP Blocks	1 / 112 (< 1 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	1 / 4 (25 %)

Figure 66 - Register Based System Summary

The under utilization of the DE10 board can provide an approach for lower end devices, however in order to fully utilize its performance advancement an iteration on the base design is undertaken.

The BRAM embedded memory (EM) bandwidth is much larger than the LE limit, having approximately 5,662 Kb available. Shown below is the Quartus Utilization Summary for the BRAM based system.

Flow Status	Successful - Thu Jun 03 22:13:08 2021
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	DE10_Standard_GHRD
Top-level Entity Name	DE10_Standard_GHRD
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	3,280 / 41,910 (8 %)
Total registers	4550
Total pins	338 / 499 (68 %)
Total virtual pins	0
Total block memory bits	2,198,656 / 5,662,720 (39 %)
Total DSP Blocks	4 / 112 (4 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	1 / 4 (25 %)

Figure 67 - Block Ram Based System Summary

Contrasting between both utilization summaries discloses the amount of BRAM utilized by the BRAM system and entails an insignificant reduction of 180 bits of logical elements. Having approximately 2195 Kbits consumption of BRAM bandwidth can be unreachable in the case of some low end FPGA devices. The BRAM iteration sports a significant decrease in execution time compared to the register based design. However, referencing table 8, the BRAM iteration would be able to house the kernel and input matrices sufficiently, it would however be unable to house the output matrix due to its sheer relative size.

In order to overcome this challenge and with the aid of the dynamic nature of our design, we designed this BRAM iteration to have the HPS process the first four convolutional blocks of our custom CNN MobileNet V1 architecture, and offload the reset of the convolutional blocks to the FPGA. This reduction provided the following largest matrix sizes metrics.

Table 10 - Largest Matrix Sizes (Reduced)

Matrix	Largest Kernel Matrix	Largest Input Matrix	Largest Output Matrix
Dimensions	256 x 9 x 16	9 x 256 x 16	256 x 256 x 32
Size (Kbit)	36.864	36.864	2,097.152

Shown in table 11 is the comparison of the register based, BRAM and the functional simulation implementation which will provide needed contrast on the whole project implementation.

Table 11 - System Implementation Metrics

DE10 Standard	Functional Simulation	Register	BRAM
Accuracy	99.7 %	99.6 %	99.6 %
Total Time (sec)	309.8	1654.3	333.5

The register based implementation is a valid approach, it is however constrained by the inefficient method of memory loading. The BRAM approach more closely resembles the functional simulation in its total time in execution. In the case of the functional simulation, the matrices are already loaded into memory and are processed on command. However, a 23.7 incurred overhead is added to the BRAM approach due to memory loading and unloading from the HPS to FPGA components.

11.8. Conclusions

Improvements on the design could find a middle ground between the register approach and the BRAM approach. Utilizing the BRAM based memory for more bandwidth, one could carry out a design as shown in table 12, where the kernel sizes remain as in the register based approach, while the input and output kernels could be adjusted accordingly.

Table 12 - Largest Matrix Sizes (Future Work)

Matrix	Largest Kernel Matrix	Largest Input Matrix	Largest Output Matrix
Dimensions	1 x 9 x 16	9 x 4096 x 16	1 x 4096 x 32
Size (Kbit)	0.144	589.824	131.072

This would introduce a lesser BRAM utilization than the approach taken in table 10, while also having all the convolutional blocks running on the FPGA.

Another method that can introduce better performance would be to treat the HPS as an API. As referenced in table 11, the time taken to complete the functional simulation is relatively long compared to other platforms. Instead of having the HPS both run the model and interface with the FPGA, we could offload the model to a faster platform and communicate with the HPS via sockets. Having the platform that is running the model simply sends the kernel and input matrices to the HPS, and having the HPS interface with the FPGA and return the output back to the device that is running the model. The overhead incurred from the socket transmission and the memory loading/unloading should be studied and accounted for.

11.9. References

[14] Andrew Ng, Kian Katanforoosh and Younes Bensouda Mourri. (2020). Deep Learning Specialization. *Deep Learning AI Courses.* Retrieved from https://www.coursera.org/specializations/deep-learning

[12] Carballo-Hernández, W., Pelcat, M. and Berry, F., 2021. *Why is FPGA-GPU Heterogeneity the Best Option for Embedded Deep Neural Networks?*. Retrieved from: https://arxiv.org/abs/2102.01343.

[15] Howard, A., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *ArXiv, abs/1704.04861*.

[17] Jin, Y. A comprehensive survey of fitness approximation in evolutionary computation. *Soft Computing* 9, 3–12 (2005). https://doi.org/10.1007/s00500-003-0328-5

[4] M. F. Uddin, "Addressing Accuracy Paradox Using Enhanced Weighted Performance Metric in Machine Learning," *2019 Sixth HCT Information Technology Trends (ITT)*, Ras Al Khaimah, United Arab Emirates, 2019, pp. 319-324, doi: 10.1109/ITT48889.2019.9075071.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. “ImageNet classification with deep convolutional neural networks”. *In Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12)*. Curran Associates Inc., Red Hook, NY, USA, 1097–1105.

[9] Kyriakos, A. (2019). High Performance Accelerator for CNN Applications. *2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)* (pp. 135-140). Athens: IEEE.