Neural Network with HLS

Karthikeyan Sugumaran
A53269850
ksugumar@eng.ucsd.edu

Imtiaz Ameerudeen
A53271622
iameerud@eng.ucsd.edu

Department of Electrical and Computer Engineering
University of California San Diego

***Abstract*— High-Level Synthesis(HLS) is an evolving technique of developing Application-Specific Integrated Circuits(ASICs) and FPGA-based hardware. Being a level of abstraction above the RTL design stage, HLS shows great potential to become the primary hardware description tool flow for quickly and efficiently architecting sophisticated logic design features and optimizing them for performance, power, and area without significant code changes. This project focuses on using HLS tools to design a popular Convolution Neural Network to analyze and understand the differences in the design process and also try to make equivalent or even better hardware with high-level synthesis compared to the traditional Verilog or VHDL-based development.**

Keywords—HLS, Neural Network, CNN, accelerator, FPGA, ASIC, machine learning

I. INTRODUCTION

High-Level Synthesis(HLS) is an evolving technique of developing Application-Specific Integrated Circuits(ASICs) and Field-Programmable Gate Array(FPGA)-based hardware solutions for several applications. As opposed to the traditional approach of developing Integrated Circuits with Hardware Description Languages(HDL), like Verilog and VHDL, HLS provides significantly faster turn-around time and a software-like hardware design environment, which makes it a viable option to implement complex algorithms with hardware logic components.

The hardware industry today is experiencing huge demands from user application industries to build devices that could accelerate machine learning workloads. There have been several architectural innovations and novel techniques to process convolutional neural networks in the past decade. While most of them have verified their claims and proposals by building actual chips, there is little emphasis on the hardware development process itself. The usage of HDLs to develop optimized hardware is a painstaking process. It involves a lot of design verification, architectural restructuring, frequent code changes, and incredible amounts of patience. Although a challenging design process, designers tend to prefer HDLs mainly because of the established development infrastructure. Electronic Design Automation(EDA) tools play a crucial role in the hardware design process and have been developed with sophisticated optimization algorithms and are constantly upgraded for these languages. Since much of the design process has been automated, the only significant portion where human interaction is unavoidable is the RTL design process. Considering the turn-around times in this stage, designers are now moving the design process a level above the RTL stage, which has led to High-Level Synthesis.

In this project, a simple neural network has been developed using HLS to explore the possibilities and capabilities of this new paradigm. The main objectives of this project are to develop a convolutional neural network library, which will contain the layers that are typically used in a CNN, to assemble a neural network by picking modules from the library, and compare synthesis results obtained from two different HLS tools, Xilinx Vivado HLS and Mentor Graphics Catapult HLS.

The rest of the report is organized in the following manner. Section II covers the basics of CNNs and describes the LeNet neural network architecture, which is developed in this project. Section III explains the architecture and implementation of each of the layers that constitute LeNet in detail and gives information on all the components built as part of the design. Section IV covers the optimization applied and undertaken with the HLS tools to increase the throughput of the design. Section V reports and compares the observed results obtained post-synthesis from both tools. Section VI concludes the report and includes future scope for this project.

II. CONVOLUTIONAL NEURAL NETWORKS

Neural Networks are a collection of smaller compute units, called neurons, that are arranged in a specific way to extract information from a given input and produce a prediction or a classification as the output. They have been widely used in the past and the industry has seen an immense increase in their popularity recently, driven by the advances in computer hardware.

Convolution Neural Networks are a subset of networks that were built to process image data. As the name suggests they use convolutions as their main form of computation as it plays a significant role in a wide range of signal and image processing applications. A very popular and one of the first successful convolution neural network was the LeNet[1] developed by Yann LeCun. It was used to recognize numbers from 0 to 9 from a given image of a digit and responds with a probability distribution that favors the digit that was given as the input.

The architecture of LeNet is depicted in Figure 1. From the image, it is evident that the net consists of several layers that move and operate on data in different ways. A 32x32, single-channel image is given as the input. The image is passed onto a convolution layer that outputs a 3-dimensional data structure of size 28x28 and 6 channels. This data structure is termed as a tensor in the machine learning domain. Convolution operations tend to extract information from a given input based on the kernel used. A kernel, in this context, is a sliding window that is moved over the image and used to perform multiply-accumulation with the image pixels. This is equivalent to applying filters over images in image processing, except here the filters are not decided by the user and are learned by the network itself through a process called backpropagation.

Following the first convolution layer, the image/input feature map size is reduced by performing pooling. There are several ways to do this and the most commonly used is the maxpool method where the maximum element of the image superposed by the kernel is retained and others are discarded. This process is performed to reduce input size and make the net extract specific patches of information from the original image. Usually, the number of kernels used in each layer keeps increasing to enable the extraction of more features. The tensor gradually becomes smaller in height and width but increases in depth. This can be observed from the image of LeNet.

The above two layers are followed by another pair of convolution and pooling layers. By the end of the second maxpooling layer, the image size becomes 5x5 but contains 16 channels. Since the end goal of this neural net is to predict the digit given as input, it must produce something as a means of expressing confidence in the digit it thinks could be(or not be) the given input image. To achieve this the 3D tensor is unwound into a linear data structure. This process is called flatting. Once flattened, the dimension of the data ends up being 400x1 in the case of LeNet.

The output of the net must be a 10-entry data structure, each representing the probability of the input being an image representing a digit from 0 to 9. To reduce the size from 400 to 10, three Fully Connected(FC) layers are used. The FC layers are named that way because every entry of the input to the layers contributes to every output of the layer. The magnitude of their contributions is represented by the weights of the layer. A dedicated set of weights are used for each output and is used by every input of the layer. Three fully connected layers are used each resulting in 120, 84 and 10 entries. Finally, the 10 entries obtained from the last fully connected layer is made to represent probabilities that add up to 1 to get the confidence of the neural net in predicting each digit for a given input. Softmax operation is used to achieve this and it is given by Eqn. 1.

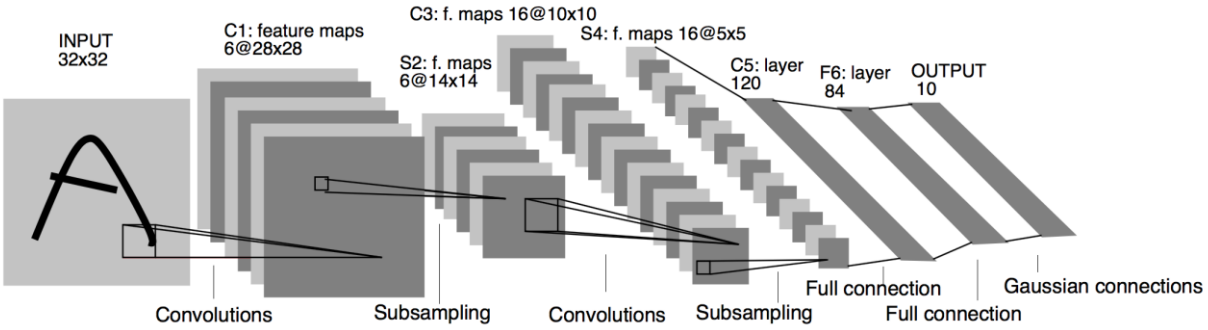

Figure 1 – LeNet Architecture

$$S(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Eqn. 1

Activation functions are used after every convolution and fully connected layer to introduce non-linearity into the learning function of the neural net. Without such activations all the computations of a neural net would boil down to one single, huge, linear vector multiplication. Introducing non-linearities in the function enable the net to learn and fit to more complex problems. The most commonly used activation function is the Rectified Linear Unit(ReLU). Relu discards any negative values in the input and sets them to zero, while maintaining all positive values as is.

This project does not go into detail about training of neural networks and assumes pre-trained weights are available. The rest of this report refers to a full computation performed by the net as an inference. Since the computations are very similar in case of a feed-forward path or training, they are not covered independently when dealing with neural network accelerators. Also, all learning-based applications only use the feed-forward path as training is just a one-time process to prepare the neural net for that application. The following section goes over the design details of each layer and the assembly of LeNet with a combination of those layers.

III. ARCHITECTURE AND IMPLEMENTATION

This section covers every layer developed as a part of this project in detail. The underlying function performed by each layer and its implementation in HLS is discussed.

A. Convolution Layer

As mentioned in the previous section, the convolution operation represents a large proportion of the neural net and a significant amount of time is spent in these layers in a feed-forward computation of any CNN. It requires two inputs, the image and the kernel. The kernel acts as a sliding window that is moved over the entire image to perform dot products with the pixels is superposes. In case of neural networks, along with the filter weights, biases are also used to the computation. Biases are standalone weights that are not multiplied with the input but are just added to the output. Therefore, the convolution operation developed in this project takes in 3 inputs and produces 1 output. Figure 2 shows a basic convolution operation performed in a CNN.

The inherent process of convolution sometimes requires the input to be padded along its boundaries. Depending on the required output size, the input is either padded with zeros or is left as it. Padding by a specific amount, which is dependent on the kernel size, results in an output that has the same size as the input or even larger than the input, which is typically not used in CNNs. Performing convolution without padding results in a smaller output image. The algorithm of convolution, as implemented in this project is shown below.

Algorithm 1 *Convolution*

Input: *image, filter, bias*

Output: *out*

Initialize with biases:

1. ***for*** *ofm=0 to ofm < out_channels* ***do***
2. ***for*** *r=0 to r < height* ***do***
3. ***for*** *c=0 to c < width* ***do***
4. *acc_buf[ofm][r][c] = bias[ofm]*

Perform sliding window operations:

5. ***for*** *ifm=0 to ifm < in_channels* ***do***
6. ***for*** *r=0 to r < height* ***do***
7. ***for*** *c=0 to c < width* ***do***
8. ***for*** *kr=0 to kr < kernel_size* ***do***
9. ***for*** *kc=0 to kc < kernel_size* ***do***
10. *pad_image(if needed);*
11. *acc+= filt[ofm[[ifm][kr][kc]*image*
12. *acc_buf += acc*

Copy accumulation to output:

13. ***for*** *r=0 to r < height* ***do***
14. ***for*** *c=0 to c < width* ***do***
15. *out = acc_buf*

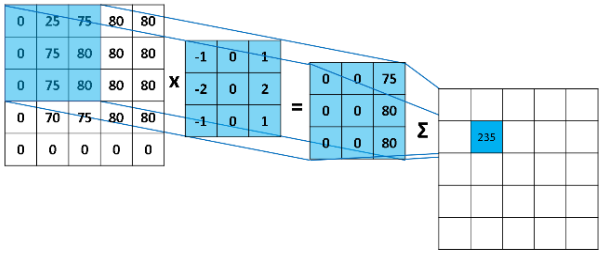

Figure 2 – Single Convolution Operation

To reduce the computation overhead of deciding if padding is required or not inside a single module, two separate functions have been developed to allow the user to choose the required type based on the neural net. This allows to have independently well-optimized designs.

The hardware architecture of convolution requires an accumulation buffer that temporarily holds multiply-accumulate values and are then copied onto the output buffer. The accumulation buffer is initialized with the bias values instead of zeros to save cycles, as they would need to be added anyway at the end of the operation. After initialization, a loop nested over the number of input channels, rows and columns of the image, rows and columns of the kernel is used to move the window over all possible location of the input and perform multiply-accumulation operations.

There are two important parameters that characterize the entire convolution operation, they are the kernel size and the stride length. The number of accumulations depends on the kernel size. Typically, a 3x3 or a 5x5 kernel is used in CNNs and this results in 9 and 25 iterations of accumulation respectively. These are accumulated onto a temporary variable and then added to the accumulation buffer initialized with bias before. The stride decides how far the kernel jumps after each set of multiply-accumulation operations. By default, the stride is set to 1 and kernel moves one element at time but increasing the stride will make the kernel move that many pixels before staring the next set of accumulations. After kernel moves to the end of the image, the contents of the accumulation buffer are copied to the output.

B. Maxpooling

The pooling follows a similar procedure as convolution where a sliding window goes over an image to perform computations. Instead of a vector product, maxpooling picks the maximum value among the values bounded by the kernel. The other commonly used kind is the average pooling where the mean of the values within the kernel window is retained. Like convolution, pooling operations also have kernel size and stride length parameters that characterize it. Since maxpooling is used in LeNet, its algorithm is shown below. Figure 3 shows a single maxpool operation.

The hardware architecture of maxpooling is a lot simpler than convolution. Since the layer only depends on the input, unlike convolution which depends on weights and biases as well, separate loops for initializing and copying are not necessary here. The algorithm involves iterating over the input image channels, rows and columns of the image, rows and columns of the kernels and the function performed within the innermost loop is the comparison of the currently stored maximum value and currently iterated over image pixel. If the image pixel is larger, the maximum value is replaced with it, else it is left as is. At the end of iterating over the kernel, the maximum value of the image region superposed by the kernel is obtained and stored as the output.

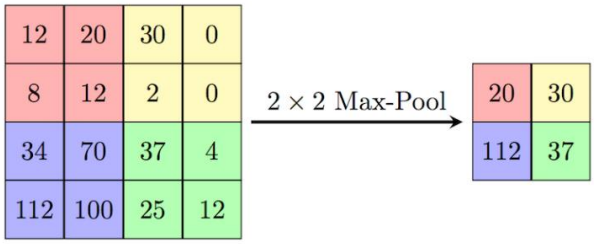

Figure 3 – Maxpool operation on single input channel

Algorithm 2 *Maxpooling*

Input: *in*

Output: *out*

Perform sliding window operations:

```
1.  max = 0;
2.  for ofm=0 to ofm < out_channels do
3.      for r=0 to r < height do
4.      for c=0 to c < width do
5.          for kr=0 to kr < kernel_size do
6.          for kc=0 to kc < kernel_size do
7.          temp = in[ofm][r + kr][c + kc];
8.              if(temp > max)
9.                  max = temp
10.     out[ofm][r/stride][c/stride] = max
11.     max = 0
```

C. Fully Connected

The fully connected layer is equivalent to performing a very large vector product. The 3-dimensional tensor that is obtained from the final pooling or convolution layer is flattened into a single-dimensional data structure. Apart from the input, like convolution, fully connected layers also require weights and biases. The algorithm is relatively simple compared to convolution, but the number of computations is equally high or even higher in case of FC layers.

Figure 4 pictorially represents how neurons are connected with each other in FC layers. The bold connections are just to denote that they are stronger than thinner ones. The strength of connection is decided by the weights they represent. A larger weight means that it gets more say in the final value of the output neuron. A combination of all such magnitudes

of the input neurons gives a single output neuron. Every output neuron is a result of a different set of weights applied on the inputs. The algorithm of FC implemented in this project is shown below

Algorithm 3 *Fully Connected*

Input: *in, weights, biases*

Output: *out*

Perform vector multiplication operations

```
1. max = 0;
2. for o=0 to o < out_length do
3.         out[o] = bias[o]
4.         for i=0 to i < in_length do
5.                 out[o] = kernel[o][i] * in[i]
```

The hardware implementation of FC is preceded by flattening the 3D tensor into a 1D array. There is no logic involved here as it is basically copying data from one memory to another in order. After flattening, the output buffer is initialized with biases like in convolution. Then each input is multiplied with the corresponding weight and then accumulated in the output buffer. Although functionally straightforward, these layers almost always become the bottleneck in the performance of several CNNs. The main reason behind this is the immensely large number of weights, bias, and corresponding multiply accumulates. The size of FC is determined by the size of tensor it receives from the previous layer. Inputs with several channels and large image sizes will result in a lot of vector product computations in FC layers.

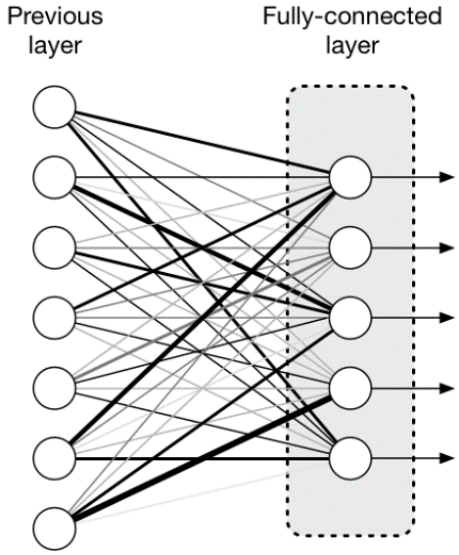

Figure 4 – Fully Connected Layer (7 → 5 neurons)

D. ReLU

The rectified linear unit activation function is simple in terms of computation. It essentially compares every entry with zero and retains if they are greater than or equal to zero and sets them to zero if they are negative. This kind of function introduces non-linearity to the inherently linear mathematical functions represented by other layers and helps the CNN learn better. The function represented by this activation function is represented by Figure 5. The algorithm implemented in HLS is shown below

Algorithm 4 *ReLU*

Input: *in*

Output: *out*

Perform comparisons to zero

```
1. for o=0 to o < out_length do
2.         for r=0 to r < height do
3.                 for c=0 to c < width do
4.                         data = in[o][r][c]
5.                         out[o][r][c] = (data > 0) ? data : 0
```

E. Softmax

Softmax is commonly used as the final layer of Neural Networks. It performs the computation shown in equation 1 on all its input elements, which is 10 in case of LeNet. The input is the output obtained from the final FC layer which transforms a data structure of length 84 to 10. Before passing this to softmax, ReLU is applied on it. The output of softmax are the probabilities predicted by the neural net that tells in which of the available classes the input could be classified into. The algorithm of softmax function is given below. This algorithm was followed for the

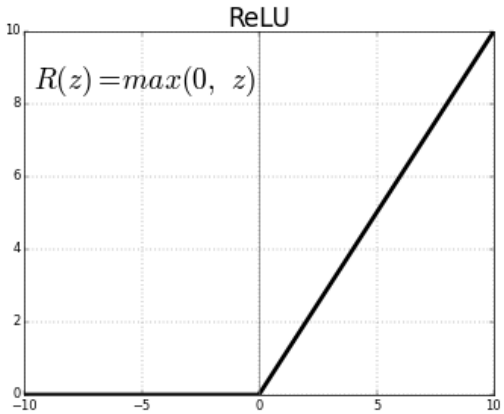

Figure 5 – ReLU Function

baseline implementation of softmax but was later modified when optimizing it and will be covered in the following section of the report.

F. LeNet Architecture

After developing all the required layers of LeNet, the neural net was ready to be assembled. It is a selected combination of the above discussed layers, sequentially arranged such that output of one layer is fed as the input to the next layer until the final 10-entry output is obtained from the last layer.

The neural net is built exactly as depicted in Figure 1. The only difference from it is the size of the input image. Instead of a 32x32 image, a 28x28 image was used as the input size. The main reason behind this is that the MNIST dataset, which was used to train the neural net, had 28x28 images. The MNIST dataset was specifically developed using a large collection of handwritten digits to train LeNet. After training with 60,000 images the net gave an accuracy of ~98.5% accuracy on the test set, which is a disjoint set of images that the net has never seen before.

A python development environment was used to perform the training and the final weights were stored to reuse it for the rest of this project. Testbenches for HLS development were built using these trained weights and biases along with the outputs obtained from the software implemented, trained LeNet.

The hardware architecture of LeNet is nothing more than the sequential arrangement of modules discussed above. The inputs and outputs of each layer were declared as arrays in HLS to allow freedom in optimizing them later. A separate header file was developed to contain all constants, parameters, and sizes of tensor across all layers of LeNet. Once assembled, the net, as a whole, was verified by giving in several samples of 28x28 images and making sure that a 10-entry softmax output was received from it.

Once the weights and biases were extracted from the software version of the neural net, it was imported onto the HLS model and now the design was verified by making sure that the outputs received predict the right digits and match the corresponding inputs. Figure 6 gives a sample input given to the net and the output obtained from it. It can be observed that the probabilities from the softmax layer of LeNet are all close to zero except for one value which is almost 1. The position of this value, starting from the 0th and ending with the 9th, is the prediction made by the net. In general, the position of the maximum value among all the output values is considered to be the prediction made by the neural net. Since, in this case, the maximum value occurs at 7th position, the digit predicted by LeNet is 7. This proves that the HLS model is functionally correct and can be optimized to make its predictions faster. The following section contains the optimizations performed on the baseline design to improve the throughput of LeNet.

IV. HLS OPTIMIZATIONS

The next step in the HLS design process is to optimize the model for throughput and area. Before beginning with improving design and source code, the baseline version's synthesis reports were recorded to later analyze the progress and speedup achieved. Since most of the development process was done with Xilinx Vivado, this section will contain the latency, area, and

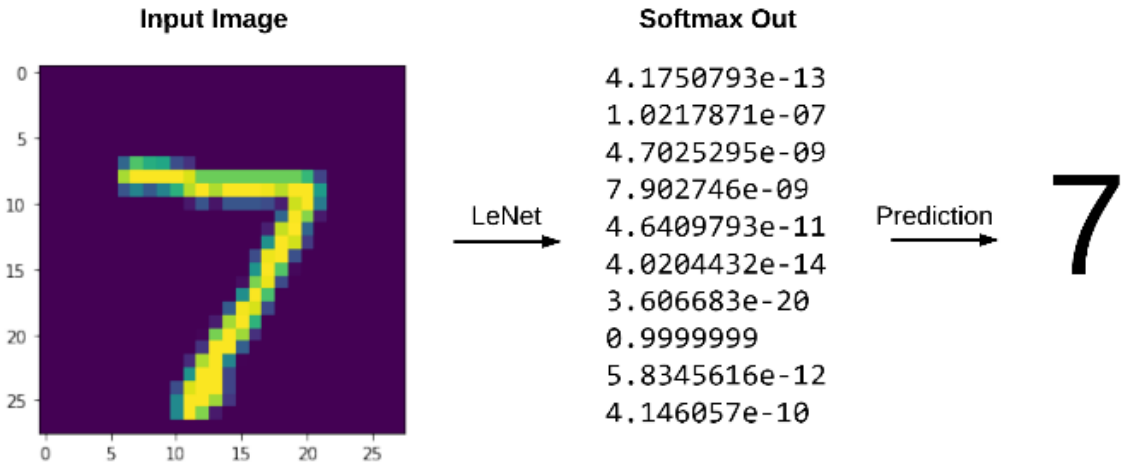

Figure 6 – Sample LeNet output

throughput numbers obtained from it. The xc7z020clg400-1 board from the Zynq-7000 family was used as the FPGA target for all synthesis experiments.

The baseline design without any optimizations consumes 4,087,010 cycles to perform a single, complete feed-forward operation. Assuming a clock time of 10ns, the throughput achieved with this implementation is approximately 24.5 Hz. This implies that the hardware can perform 24.5 inferences or predictions per second. Figure 7 below shows the latency numbers for every layer, more specifically, the latencies of all instances used in the baseline design. It is important here to note that almost 70% of the runtime comes from a single convolution layer, which is denoted by the module conv2d_C2. This is followed by the other convolution layer(C1) and the FC layers. A critical factor to analyze before optimizing any design is to decide on which components that constitute the design could result in the greatest improvements.

Figure 8 shows the resource utilization for the baseline design and figure 9 shows layer-wise resource utilization to get more intuition about the distribution of the resources. It can be observed that almost 80% percent of the BRAMs are used to hold weights, biases, and the intermediate tensors of every layer. The observed area numbers seem to be considerably higher for an unoptimized design with a very low throughput. The optimizations performed must use hardware effectively and efficiently even if a lot more resources are required. The rest of this section will cover the layer-wise optimizations performed to improve the throughput of the baseline design. Vivado HLS provides with a wide range of directives that can be applied over loops, functions, and memories to enable smooth dataflow and effective utilization of the hardware.

Instance

		Latency		Interval		
Instance	Module	min	max	min	max	Type
softmax_SM_U0	softmax_SM	322	322	322	322	none
conv2d_C2_U0	conv2d_C2	2830033	2830033	2830033	2830033	none
conv2d_C1_U0	conv2d_C1	551407	551407	551407	551407	none
fc_FC1_U0	fc_FC1	528361	528361	528361	528361	none
fc_FC2_U0	fc_FC2	111133	111133	111133	111133	none
fc_FC3_U0	fc_FC3	9271	9271	9271	9271	none
maxpool_P1_U0	maxpool_P1	21349	21349	21349	21349	none
maxpool_P2_U0	maxpool_P2	7393	7393	7393	7393	none
relu_R1_U0	relu_R1	19165	19165	19165	19165	none
relu_R2_U0	relu_R2	6753	6753	6753	6753	none
relu_R3_U0	relu_R3	481	481	481	481	none
relu_R4_U0	relu_R4	337	337	337	337	none
flatten_F_U0	flatten_F	993	993	993	993	none

Figure 7 – Timing report of Baseline design

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2	-
FIFO	-	-	-	-	-
Instance	177	30	6075	11973	0
Memory	50	-	64	5	0
Multiplexer	-	-	-	-	-
Register	-	-	-	-	-
Total	227	30	6139	11980	0
Available	280	220	106400	53200	0
Utilization (%)	81	13	5	22	0

Figure 8 – Total Resource Utilization of Baseline design

Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT	URAM
conv2d_C1_U0	conv2d_C1	3	5	846	1591	0
conv2d_C2_U0	conv2d_C2	9	5	903	1701	0
fc_FC1_U0	fc_FC1	129	5	562	920	0
fc_FC2_U0	fc_FC2	33	5	538	905	0
fc_FC3_U0	fc_FC3	2	5	569	901	0
flatten_F_U0	flatten_F	0	0	91	269	0
maxpool_P1_U0	maxpool_P1	0	0	234	717	0
maxpool_P2_U0	maxpool_P2	0	0	234	701	0
relu_R1_U0	relu_R1	0	0	198	523	0
relu_R2_U0	relu_R2	0	0	195	507	0
relu_R3_U0	relu_R3	0	0	157	383	0
relu_R4_U0	relu_R4	0	0	157	383	0
softmax_SM_U0	softmax_SM	2	5	1391	2472	0
Total	13	178	30	6075	11973	0

Figure 9 – Instance-wise Resource Utilization of Baseline design

A. Convolution

We observed that convolution layers take up more than 70 percent of the total runtime of the entire network. On larger CNNs they consume about 90-95% of the total runtime. Optimizing them is the key to achieve great performance gains when accelerating neural networks. The convolution layer, as explained in the previous section of this report consists of three loops. All the loops that iterate over columns were pipelined to start the next iteration as soon as possible. Pipelining the row loops, which are a level higher than columns, results in unrolling all the columns and a significantly long synthesis runtime. The results obtained were not very impressive as they indicated overutilization of resources for very low improvement in latency. Therefore, only the column loops in all the 3 major loop sections were pipelined.

Following pipelining, the throughput of convolution was further improved by loop unrolling. Unrolling the

convolution operation was not as straightforward as adding a pragma over the region to unroll. Instead, the loops were manually unrolled for a couple of reasons, (i) the convolution operation does not access elements sequentially from the memory (ii) manually unrolling gives more control on which aspect of the algorithm to parallelize. Since the convolution operation provides with several tiers of potential parallelism, choosing the wrong one could hinder the parallelism of the others. Initially, the innermost loops that iterate over the kernel window was completely unrolled. The directly was not explicitly mentioned as pipelining the loop above them unrolled them automatically. After this the innermost loops of all the three loop sections were manually unrolled 4 times, where each statement of the unroll accumulates a different out channel but reads the same input pixels.

To complement the unrolled loops, the arrays that needed to be accessed in parallel were partitioned by a factor of 4 with cyclic mode. This made each statement of the unrolled section to access its required data simultaneously and moreover, also write to the output array simultaneously after the computations are performed. This way both parallelism and data reuse were exploited to give almost 4x speedup over the pipelined version. An overall speedup of almost 87x was achieved with the combination of all the above optimizations. Similar optimizations were applied on the other convolution layer, but the loops were unrolled twice as the number of output channels were lesser in this case.

B. Fully Connected

The fully connected layers are the next most time-consuming components of LeNet. This intuitively simple operation consumes a surprisingly large number of cycles due to the sizes of the data structures used. It also involves reading weights and biases adding on the latency for each computation. The FC layers were optimized the same way as the convolution layers with pipelining, unrolling, and partitioning. Pipelining the inner loops ensured computations are performed as soon as the previous ones are completed and promoted constant utilization of the available hardware. Since FC layer is equivalent to several dot product operations, unrolling the loop made several data points to be worked on at once. This concept is very similar to SIMD form of computation in traditional, general purpose processors and GPUs. The same operation of multiplication and accumulation is performed on every data point in parallel. The arrays that required parallel reads and writes were partitioned with factor equal to the number of times the loop was unrolled. The largest FC layer was unrolled 8 times, while the successive, smaller ones were unrolled 4 times and 2 times respectively. An overall speedup of ~17.5x was achieved on the largest FC layer.

C. Maxpool and ReLU

The maxpool layer is similar to the sliding window method used by convolution but not as complex. ReLU is even simpler with fewer comparisons than maxpool. These layers inherently consume a lot lesser number of cycles than convolution or FC. The sliding window kernel in maxpool was automatically unrolled when the parent loop was pipelined. To save hardware, unrolling was not used for ReLU layers and only pipelining was incorporated. They both resulted in about ~8-9x speedup over the baseline. Optimizing them further at the cost of more hardware did not make sense as the throughput of the entire net will depend on the component that consumes the most time. More details on this will be covered later.

D. Softmax

Although, softmax might seem like a simple set of operations over a small data structure, it involves performing exponential operations that requires Vivado's math library. Directly using the exponential function from the library in the HLS code resulted in a lot of unnecessary cycles that seemed wasteful for a simple layer that is going to be used only once per inference. To correct this, the exponential values of a range of values that the softmax input could take were precalculated and stored in the memory to act as a lookup table whenever the exponential of a value is required. A resolution of 400 was used, which means the range was split to contain 400 values within the lowest and highest exponential. This resulted in a reduction in the number of DSPs used and also a reduction in the number of cycles to complete the computations of the layer.

E. LeNet

After all the layers were optimized individually, they were assembled together to form an optimized neural network. Since LeNet does not contain any extra logic by itself, no more optimizations were used except dataflow. Vivado HLS provides with a pragma that enables task-level parallelism and pipelining. Considering each layer to be a task, once a task is completed by the hardware dedicated for it, another task of the same kind can be computed with that hardware. This is the basis of pipelining in any computer and neural nets can be exploited with such an optimization as there is dataflow in a single

direction and each function is unique and used only once per inference. Applying this optimization improves throughput significantly and allows to achieve larger inferences per second which is crucial for real-time image recognition CNNs.

After all the above optimizations were applied, the synthesis resulted in a latency of 101,317 cycles, which is almost 40x faster than the original latency of 4,087,010 cycles. Figure 10 shows the latencies obtained for all the layers in the optimized design. This is the speedup achieved for one complete feed-forward process of LeNet without considering task-level pipelining. After applying the dataflow optimization, the latency remained the same but the interval after which the processing of a new image could began dropped to 32,534 cycles resulting in a throughput of 3.073kHz. From this task-level parallelism, it can be observed that the overall speedup achieved over the baseline is approximately 125x. This means the optimized design can perform almost 3000 inferences per second as opposed to 24 inferences that the baseline was able to perform.

The resource utilization of the optimized LeNet is shown in Figure 11 and the layer-wise resource utilization is shown in Figure 12. Although it is a lot higher than what the baseline used, it is very reasonable for the 125x speedup that was achieved. The number of BRAMs remain almost the same, while the other resources have increased significantly. This is mainly due to pipelining and unrolling which require a lot of registers and extra logic components to perform computations in parallel. Considering the total number of all kinds of hardware components, the baseline version uses 18,376 area units while the optimized version uses 73,849 area units. This implies that the optimized version uses almost 4 times that area used by the baseline version. This can be attributed to

Instance

		Latency		Interval		
Instance	Module	min	max	min	max	Type
conv2d_C2_U0	conv2d_C2	32533	32533	32533	32533	none
conv2d_C1_U0	conv2d_C1	16663	16663	16663	16663	none
relu_R1_U0	relu_R1	2358	2358	2358	2358	none
fc_FC1_U0	fc_FC1	30196	30196	30196	30196	none
softmax_SM_U0	softmax_SM	88	88	88	88	none
relu_R2_U0	relu_R2	806	806	806	806	none
fc_FC2_U0	fc_FC2	12874	12874	12874	12874	none
maxpool_P1_U0	maxpool_P1	2363	2363	2363	2363	none
maxpool_P1_1_U0	maxpool_P1_1	811	811	811	811	none
fc_FC3_U0	fc_FC3	2156	2156	2156	2156	none
relu_R3_U0	relu_R3	125	125	125	125	none
relu_R4_U0	relu_R4	89	89	89	89	none
flatten_F_U0	flatten_F	243	243	243	243	none

Figure 10 – Timing report of Optimized design

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	102	-
FIFO	-	-	-	-	-
Instance	172	85	36259	35967	0
Memory	54	-	896	114	0
Multiplexer	-	-	-	180	-
Register	-	-	20	-	-
Total	226	85	37175	36363	0
Available	280	220	106400	53200	0
Utilization (%)	80	38	34	68	0

Figure 11 – Total Resource Utilization of Optimized design

Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT	URAM
conv2d_C1_U0	conv2d_C1	4	20	5553	5197	0
conv2d_C2_U0	conv2d_C2	4	40	20774	13071	0
fc_FC1_U0	fc_FC1	128	10	2156	2162	0
fc_FC2_U0	fc_FC2	32	5	1154	1182	0
fc_FC3_U0	fc_FC3	2	5	813	1051	0
flatten_F_U0	flatten_F	0	0	100	478	0
maxpool_P1_U0	maxpool_P1	0	0	783	1538	0
maxpool_P1_1_U0	maxpool_P1_1	0	0	750	1472	0
relu_R1_U0	relu_R1	0	0	1248	4191	0
relu_R2_U0	relu_R2	0	0	721	2115	0
relu_R3_U0	relu_R3	0	0	314	487	0
relu_R4_U0	relu_R4	0	0	314	487	0
softmax_SM_U0	softmax_SM	2	5	1579	2536	0
Total		13	172	85	36259	35967	0

Figure 12 – Instance-wise Resource Utilization of Optimized design

the large number of extra registers used for pipelining, more LUTs and BRAMs due to array partitioning, and a significantly higher number of DSPs due to unrolling and parallelism exploited in computations.

Modern neural networks have been handcrafted with the right number of bits for each kind of data it uses. This is usually done to make processing them faster with hardware. On top of all the optimizations discussed previously in this section, bitwidth optimizations for the different kinds of data used in the design have also been incorporated. The drawback, although acceptable depending on the application of the neural network, is that reducing bitwidths of datatypes will also reduce the range of data that can be held by the variables in the neural net. This affects the overall prediction accuracy and also the variety of applications it can be used for. Accuracy critical applications like autonomous driving and healthcare cannot compromise the correctness of the model for performance. This makes variable bitwidth

optimizations very important and requires shrewd decisions from hardware designers. A small experiment to analyze how bitwidths affect accuracy, performance and area of LeNet has been conducted and is discussed in section VI. The following section will discuss about the porting the LeNet design from Vivado to Catapult and the optimizations there were performed on the baseline design with Catapult-specific directives. A comparative study between the two tools has also been made.

V. VIVADO VS CATAPULT

Mentor Graphics' Catapult HLS was also used for optimizing and synthesizing the baseline design. Though the optimizations and coding style for both the tools are somewhat similar, the synthesis itself might not give the same results. Catapult, in this project, was used to analyze how LeNet gets synthesized as an ASIC rather than an FPGA. The target technology used was the Catapult default, nangate-45nm and the synthesis tool flow was OasysRTL. The target timing conditions were a clock period of 10.0ns and a clock uncertainty of 1.25 ns, which is the same as the constraints used for synthesis in Vivado. The rest of this section covers the optimizations applied on LeNet using Catapult, the results reported, and a comparison with the results from Vivado.

The baseline design from Vivado was ported to Catapult after removing all optimization pragmas from before to allow Catapult-specific directives. This unoptimized version resulted in a latency of 1,516,630 cycles, which is equivalent to a throughput of 65.9Hz. This is considerably higher than the baseline from Vivado because of the targeted hardware. Since Catapult's target is an ASIC with 45nm technology node, it exploits the freedom of using the given component library to build an optimized design that falls within the timing constraints set by the user. Also, since Vivado synthesizes design for an FPGA there are certain inherent design and hardware component constraints to achieve a better latency number. The runtime profile for the baseline design from Catapult is show in Figure 13. It can be observed that almost 65-70% percent of the runtime is used up by the second convolution layer(C2_OFM) similar to the report obtained from Vivado. The other layers that consume a large percentage of total runtime are the first convolution layer(C1_OFM), the first and second FC layers denoted by FC1_OUT and FC2_OUT respectively in the figure.

The first step in the Catapult optimization phase was to apply pipelining on promising loops of the design. An interesting observation here was that the initiation interval(II) of certain loops that were achieved without any scheduling errors in Catapult, were not the same as they were automatically achieved in Vivado. The reason behind this could be scheduling algorithm of the tools, which tries to find possible solutions for the problem using available hardware resources and timing constraints to synthesize a working design. Due to this, the way hardware components are scheduled for a given function could affect when the next iteration of the same function could begin. Table I shows the II achieved in both tools for all the loops pipelining was applied.

After pipelining, the next step performed was loop unrolling. All the loops that involved the usage of kernels, which are the innermost loops of the convolution and maxpool layers, were completely unrolled. Unlike Vivado, which automatically unrolled these loops when the parent loop was

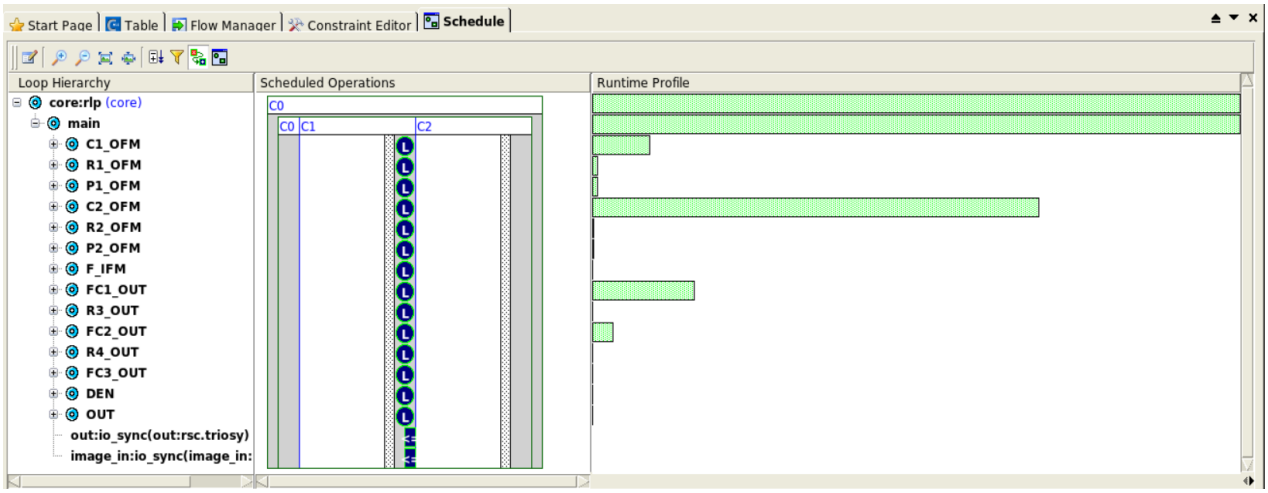

Figure 13 – Runtime profile of Baseline design

pipelined, Catapult required the user's decision in unrolling loops within pipelined sections of the code.

Table I – Layer-wise Initiation Intervals

Layer	Loop	Vivado (II)	Catapult (II)
Convolution with padding, kernel size 3	Bias	1	1
	Sliding Window	5	10
	Out	1	1
Convolution w/o padding, kernel size 5	Bias	1	1
	Sliding Window	13	26
	Out	1	1
FC	Dot Product	5	2
Maxpool, window size 2	Sliding Window	2	4
Relu	Compare	1	1
Softmax	Denominator	5	1
	Out	1	1

The bottleneck after the above optimizations were the memory accesses. Similar to the Vivado implementation, the memories that needed to be accessed in parallel were partitioned to allow multiple reads and writes at the same time. After this a latency of 126,786 cycles was achieved for one complete feed-forward path of LeNet. The throughput of the design is dependent on the layer that consumes the largest number of cycles. In this case, the second convolution layer took 66,548 cycles to complete its computations. Therefore, the achieved throughput of the design was 1.50kHz. This is approximately half of the throughput achieved with Vivado.

Table II shows the layer-wise latencies achieved using both the tools for the optimized designs. The results obtained here seem interesting as both tools are optimizing different layers of the neural network differently. Vivado performs well in optimizing Convolution, pooling, and relu layers to achieve a higher throughput that its Catapult counterpart. It is difficult to make intuitive conclusions about why this is the case as the tool optimization algorithms are not disclosed to end-users. A possible explanation could be that unrolling and partitioning are well executed in Vivado than Catapult as convolution and maxpooling both use the sliding window method, which reading the memory efficiently and in parallel important in this approach. One the other hand, Catapult performs better on FC layers than Vivado with almost 2x-3x speedup. Since the FC layers are inherently large number of vector multiplications, pipelining might be a more important optimization here. Since Catapult achieves an II of 2, which is 3 cycles lower than Vivado's II of 5 as observed from Table I, it performs noticeably better on fully connected layers. It can also be noted that the II achieved by Catapult on the convolution layers are significantly higher than the values achieved by Vivado, which could mean that the memory reads are the bottleneck here although similar combination of loop unrolling and array partitioning have been incorporated.

Table II – Layer-wise latency report

Layer	Vivado(cycles)	Catapult(cycles)
Conv1	16663	29067
Conv2	32533	66548
FC1	30196	12105
FC2	12874	5166
FC3	2156	860
Maxpool1	2363	4962
Maxpool2	811	1856
Relu1	2358	4722
Relu2	806	1648
Relu3	125	122
Relu4	89	86
Softmax	88	37

The runtime profile for the optimized design from Catapult is show in Figure 14. The graph has changed considerably from the one obtained from the baseline design. Although the same layers, Convolutions and FC, take a large percentage of the total runtime, they have now become comparable to other layers as they are optimized and consume significantly lesser number of cycles. The other layers like Pooling and ReLU show up on the radar unlike the baseline design, which proves that all layers have been optimized and the throughput is more equally distributed than before which is vital for achieving large throughputs.

Finally, the area difference from the baseline and the optimized versions of LeNet in Catapult is proportional and similar to the difference obtained in Vivado. The baseline design utilizes an area of 5,943,135 area units while the optimized design uses 39,225,529 area units. The optimized design uses approximately 6.6x the area used by the baseline design. This is expected given the amount of pipelining, parallelism through loop unrolling, and array partitioning incorporated to increase the throughput of the design. It can be also noted that Catapult used more resources than Vivado, which had 4x area increase, in their optimized versions.

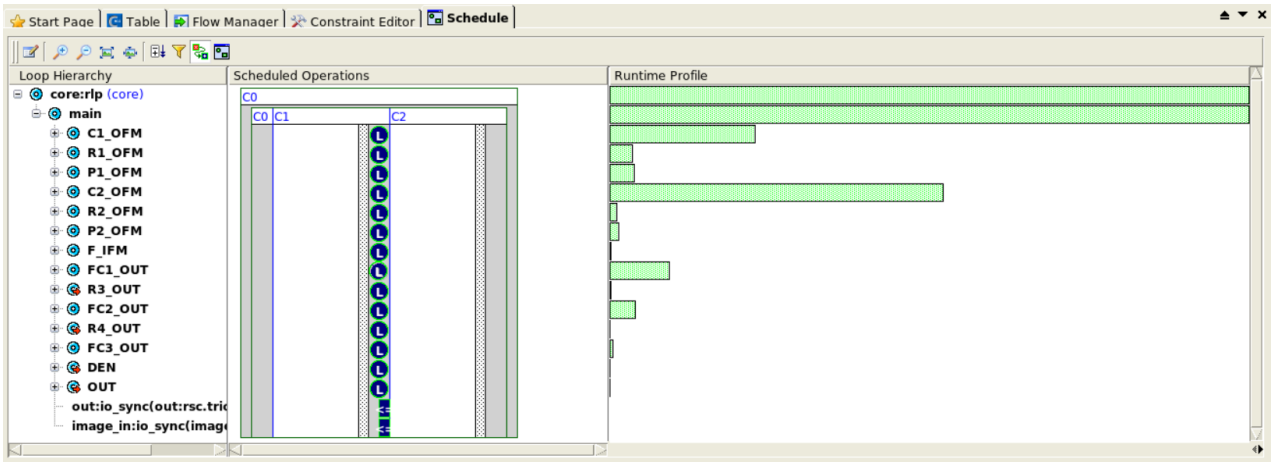

Figure 14 – Runtime profile of Optimized design

VI. EXPERIMENTS

A. Variable Bitwidths

As discussed in Section IV, a brief experiment with variable bitwidths was performed on the optimized LeNet model. The 3 main datatypes used in the design are DTYPE, PARAM_TYPE, and EXP_TYPE. DTYPE corresponds to the datatype that is used for all input, output and intermediate data structures. In other words, this is the datatype that characterizes the image input, the dataflow along the layers, and the final output of the neural net. PARAM_TYPE is used to represent all the weights and biases of the model. These are static datapoints and, unlike DTYPE, may not require extensive analysis to find the best datatype that will not result in wrong predictions. Finally, EXP_TYPE datatype is used exclusively for the lookup table used in softmax layer. Since this is also static and does not change with runtime, a quick analysis of the range of values present in the lookup table will lead us to the optimal datatype.

Table III shows the datatypes tried on the optimized LeNet design. The accuracy of every datatype combination shows in the table was obtained by iterating over 500 images from MNIST's test set in LeNet's testbench. The latency and area numbers were also recorded.

It can be observed that using floats or fixed-point datatypes with 64-bit words and 32-bit integral part give an accuracy of 99.2%. This is equivalent to the accuracy achieved with the TensorFlow implementation of LeNet evaluated using the same 500 image test set. Although they provide with the same accuracy, they give very different latency and area values after synthesis. The implementation using floats consumes ~100k cycles while the one with fixed-point variables takes ~76k cycles. This is the latency to complete one complete feed-forward path of the neural net and does not reflect the throughput of the design which depends on the initiation interval between two different feed-forward paths. There was little to no difference in the throughput and both the implementations achieved a throughput of 3.1kHz. Area differences are significantly large between the two implementations. The fixed-point design uses more of every kind of resource with almost 11 times more DSPs. As evident from the table, this improvement in latency for the amount of resources used is not a fair tradeoff.

The rest of the table contains several combinations of fixed-point types for the 3 major datatypes discussed before. A fixed-point datatype with 32-bit word length and 2-bit integral part was used for parameters as the range of values represented by the weights and biases were covered by it and resulted in the same accuracy. As the data and exponential bitwidths were lowered, the accuracy started decreasing. A 94% accuracy is achieved with 48 bits, 88% with 44 bits, and 85% with 42 bits. Accuracy dropped significantly when 32-bit words were used for data and exponentials. Almost all the cases discussed above had higher resource utilization while giving very small latency improvements compared to the implementation with floating point variables and therefore, floating points were used for the optimized design and demo.

Table III – Variable bitwidth optimizations with accuracy and synthesis reports

Datatype Combination	Accuracy (%)	Latency (cycles)	BRAM	DSP	FF	LUT
Floating Point	99.2	101317	226	85	37175	36363
ap_fixed<64,32>	99.2	76528	296	928	105254	68736
exp: ap_fixed<64,32>, data: ap_fixed<64,32>, param: ap_fixed<32,2>	99.2	76524	288	928	104192	68396
exp: ap_fixed<48,24>, data: ap_fixed<48,24>, param: ap_fixed<32,2>	94	70409	253	792	67380	42527
exp: ap_fixed<44,22>, data: ap_fixed<44,22>, param: ap_fixed<32,2>	88.4	70427	245	792	66217	52207
exp: ap_fixed<42,21>, data: ap_fixed<42,21>, param: ap_fixed<32,2>	85.4	70409	244	528	62017	41735
exp: ap_fixed<32,16>, data: ap_fixed<64,32>, param: ap_fixed<64,32>	76.6	76524	286	928	104116	68300
exp: ap_fixed<32,16>, data: ap_fixed<48,24>, param: ap_fixed<32,2>	76.6	70409	253	792	67380	42527
exp: ap_fixed<32,16>, data: ap_fixed<48,24>, param: ap_fixed<32,2>	54	70409	253	792	67376	42526
exp: ap_fixed<48,24>, data: ap_fixed<32,8>, param: ap_fixed<32,2>	38.4	70387	219	132	32426	31593

B. Demo

The demo part of this project involves generating the bitstream of the optimized design with AXI4 interfaces for input and output ports of the model. The bitstream files were imported to PYNQ Z2 board for the demo. The Jupyter notebook included with this project contains several image samples from the test set and the predictions given out by the board to verify if the design has been correctly mapped onto the FPGA. Moreover, runtime comparison between the FPGA implementation and the default CPU implementation of test set evaluation was carried out. The time taken for evaluating 10000 images of MNIST test set was measured in both cases. The PYNQ board took 0.397 seconds while the host machine, with Intel Core-i7 8750H processor, took 1.05 seconds for the same task. Although not a highly accurate measurement of the runtimes, it can be safely estimated that the FPGA implementation is at least 2.6x faster than a high-end 6-core CPU. This validates the optimizations performed on the neural network and also bolsters the idea of using HLS tools to design hardware.

VII. CONCLUSION AND FUTURE SCOPE

HLS tools provide the semiconductor industry with a designer-friendly platform to begin the hardware development process that complements architecture exploration and relative faster feedback that could immensely speedup the hardware design cycle. In this project, a neural network library with the most commonly used layers and the LeNet convolutional neural network were developed using Xilinx Vivado HLS and Mentor Graphics Catapult HLS. The design

was functionally verified and was optimized independently with both tools using their tool-specific optimization options and the synthesis reports were compared. A design space exploration by varying the bitwidths of the datatypes was performed and the most optimal design was programmed on a PYNQ Z2 board to showcase its working and runtime improvement over a general-purpose CPU.

There are several directions to go forward from this project. Since the project concentrated more on layer-wise design and overall optimization exploration using two HLS tools, a simple CNN was used as the core design. Modern CNNs, more sophisticated and complicated than the LeNet could be build and optimized for a target FPGA using HLS. This would also raise an interesting problem of find the right FPGA target for a given neural network based on the weights, biases and other static data that come along with it. Also, throughout the course of this project, the weights and biases of LeNet were assumed to be present in on-chip memories. This is impractical with larger CNNs and efficient data movement becomes key when the weights are stored outside the chip as it could lead to huge loss in performance when not used properly. A detailed study of how storage of weights would work in FPGA and ASIC-based accelerators for neural networks would be a great avenue to further exploit the capabilities of HLS. Memory controllers need to be used when the weights are stored off-chip and dedicated control logic must stage the required data inside the chip, onto a local, cache-like scratchpad for faster accessing. This has to be done while the previous layer is computed to ensure that the throughput is not largely affected by off-chip memory accesses.

The neural network library developed in this project does not cover all possible layers that modern neural networks use. Another future scope could be to develop and optimize other components that are used in deep learning. There are more activation layers like hyperbolic tangent, sigmoid, and leaky ReLU and more kinds of pooling such as average pooling, minpooling which, although functionally very similar to the current implementation, could be part of the NN library to make it cover a wider range of neural networks.

ACKNOWLEDGMENT

We thank Prof. Ryan Kastner of University of California San Diego for motivating and guiding us throughout this project. We also thank Sivasankar Palaniappan, the instructional assistant for the course Validation and Testing of Embedded Systems, for helping us throughout the course and during the initial stages of this project. We would also like to extend our gratitude to Mike Fingeroff, HLS Technologist at Mentor, who introduced us to Catapult tool flow and helped us with the project description. Finally, we thank UCSD for giving us the opportunity to work on this project as a part of this course and our classmates for providing insights and suggestions to improve it.

REFERENCES

[1] LeCun, Yann. "LeNet-5, convolutional neural networks." *URL: http://yann. lecun. com/exdb/lenet* 20 (2015): 5.