Acceleration of AlexNet for ImageNet Classification

Dhruva Dilip Devasthale, Prabhleen Kaur Gill, Vandita Shetty

***Abstract*— AlexNet is a CNN architecture designed by Alex Krizhevsky in collaboration with Ilya Sutskever and Geoffrey Hinton [1]. The original paper's primary result was that the depth of the model was essential for its high performance, which was computationally expensive, but made feasible due to the utilization of graphics processing units (GPUs) during training. AlexNet is an incredibly powerful model capable of achieving high accuracies on very challenging datasets. Looking at these points, the aim of our project is to: 1) Write a C++ implementation for AlexNet algorithm. 2) Write an HLS implementation for the same using Vitis HLS. 3) Optimize the operation so as to perform the computationally intensive task using minimal resources in less amount of time on FPGA.**

I. INTRODUCTION

AlexNet is a convolutional neural network (CNN) architecture with 60 million parameters and 650,000 neurons, consisting of five convolutions layers and three fully connected layers with a final 1000-way softmax.

A. Architecture

The architecture [Fig. 1.] contains eight layers; five convolutional layers followed by three fully connected layers [1]. Response normalization layers are used after the first and second convolutional layers. Max pooling layers follow both response normalization layers as well as fifth convolutional layer. Further, ReLU activation function is used as the activation function after every convolutional and fully connected layer.

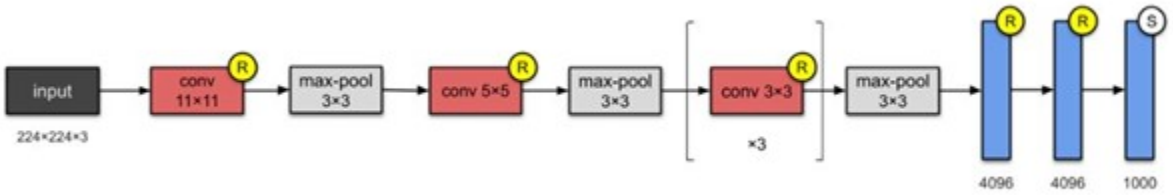

Fig. 1. AlexNet Architecture

The first convolutional layer takes an input image of size 224 x 224 x 3 with 96 kernels, each of size 11 x 11 x 3 with stride of 4 pixels. Stride is defined here as the distance between the receptive field centres of neighbouring neurons in a kernel map. The second convolutional layer takes as input the response normalized and max pooled output of the first convolutional layer and uses 256 kernels of size 5 x 5 x 48 each. The third, fourth and fifth convolutional layers are connected one after the other without any intervening. The output of the second layer is response normalized and max pooled to give it as input to the third convolutional layer which has 384 kernels, each of size 3 x 3 x 256. The fourth convolutional layer has 384 kernels of size 3 x 3 x 192. The last convolutional layer, fifth layers has 256 kernels of size 3 x 3 x 192. Lastly, all the fully connected layers have 4096 neurons each.

B. Local Response Normalization

Local response normalization was introduced for the very first time in AlexNet. It is a type of lateral inhibition in which a neuron's response to a stimulus is inhibited by the excitation of a neighbouring neuron. In AlexNet the neuron with maximum pixel has the maximum effect in the excitation of the following layer. This phenomenon was termed as "brightness normalization" in the original paper. It is along the similar lines to local contrast scheme of Jarrett et al. [2].

The response normalization activity b_x, y^i, where i is the i^th kernel at position (x, y) with activity a_x, y^i (activity is computed by applying ReLU nonlinearity) is given by

$$b_x, y^i = a_x, y^i / (k + \alpha \sum_{j=max(0,i-n/2)}^{min(N-1,i+n/2)} (a_x, y^j)^2)^\beta$$

Where the sum runs over n "adjacent" kernel maps at the same spatial position, and N is the total number of kernels in the layer.

C. Overlapping Pooling

The architecture proposed in the original paper uses overlapping max pooling. Max pooling is a pooling operation that selects the maximum element from the region of the feature map covered by the filter. Therefore, the output after max-pooling layer would be a feature map containing the most prominent features of the previous feature map. This is needed since the output of one layer which will be the input of next layer needs to be resized using max pooling. Traditional max pooling methods did not see any overlap. To be more precise, a pooling layer can be described as a grid of pooling units spaced s pixels apart, each summarizing

neighbourhood of size zxz. If we use $s = z$, we will obtain a traditional pooling method with no overlapping. But in the AlexNet architecture overlapping pooling was proposed, where $s < z$ was used, meaning there is going to be sharing of information within the neighbours. This ensures less loss of information between two layers. In this architecture we are going to follow the original paper and use $s = 2$ and $z = 3$.

II. Problem Description

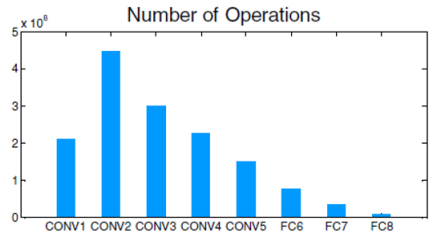

Fig. 2. Number of operation in each layer

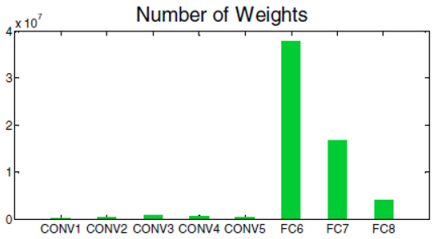

Fig. 3. Memory requirement in each layer

Given the detailed architecture of AlexNet in the previous section, we know it is computation-intensive and memory-intensive due to its multi layer structure consisting convolutional layers, response normalization, pooling and fully connected layers.

Figure 1 shows the number of operations in each convolutional and fully connected layer of the architecture. As exptected, the number of operations are high in the convolutional layers, with conv2 having the maximum number of operations due to its larger kernel size and stride of just 1. Therefore, our focus for the convolutional layers was to parallelize the computation operations by using more resources and eventually bringing down the latency.

Figure 2 shows the memory requirement of each convolutional and fully connected layer of the architecture. As expected, the memory requirement of fully connected layers is very large due to its dense nature and large dimensions of weights. Out of all layers FC1 had the largest memory requirement due to it's weight dimension of 9216x4096. Therefore, our primary focus for the fully connected layers was to bring down the BRAM usage within board limit.

During our implementation, we were limited by resources available to us on the board. Therefore, our primary objective while doing HLS was to first bring down the resource utilization within limits of the resources available on the board and then optimize the latency.

III. Implementation

A. Python Implementation

AlexNet architecture was successfully implemented in Python using Tensorflow and Keras. Dataset used was CIFAR-10, created by Alex Krizhevsky himself. The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. These input images were augmented to 227x227 and given as input to the AlexNet model.

Keras sequential model was used for building all the convolutional and fully connected (FC) layers of the architecture. All layers were kept consistent with the original architecture proposed by Alex Krizhevsky, except for the normalization layers. Here we used batch normalization instead of local response normalization. Reason being, recent studies and trends in the machine learning domain show batch normalization gives better results, and is the current standard in this domain. Moreover, local response normalization function has been removed from the Keras library.

Finally, the model was trained using the dataset with 50 epochs and standard gradient decent (SDG) as the optimizer giving a final accuracy of 90% on the testing data. All the weights and biases trained in our model with 90% accuracy were saved and used for the next step, C implementation.

B. C++ Implementation

Using the same dimensions for all the layers as in the Python implementation, Alexnet was successfully implemented using C++. This has been done to start with a correct logical implementation of the algorithm and move forward with the HLS implementation having the C implementation as a base.

The C++ implementation was done by implementing a separate C++ executable for each of the convolution, normalization, pooling and fully connected layers. A separate testbench was created to verify the functionality of each layer. Weights and biases obtained from the Python model were used for this. Output maps generated by the Python model were used as "golden outputs" to verify the correct functionality of each layer separately. At the end, a wrapper testbench was created to connect all the layers in the Alexnet implementation and verify the classification algorithm as a whole.

As mentioned in the previous section, we have used Batch Normalization in the Python implementation for generating the weights and biases to be used for these layers. But, to be consistent with the original Alexnet implementation, we have used local response normalization in the C++ implementation. By examining the difference in magnitude of normalization done in the two different algorithms, we modified the α in the local response normalization in order to generate a similar effect.

For verifying the logic, we tested how correctly the algorithm classifies an image. The input image fed belongs to Class 4. The output that our implementation generated is shown in Fig 1. The printed values are - probability of the image belonging to each class from 1 - 10. As we can see the algorithm correctly predicts, with a probability of 0.9267 that the input image belongs to Class 4.

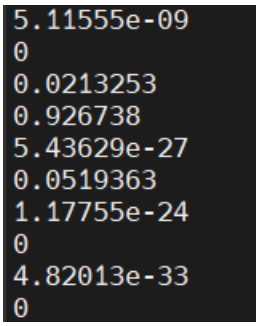

Fig. 4. Probability of input image belonging to one of the Class labels 1-10

C. HLS Implementation and acceleration

After successfully doing the C++ implementation of the architecture, we did the HLS implementation in four steps: 1) Started by implementing each of the layer separately without any optimizations. 2) Optimized all layers separately to bring down the resource usage and minimize latency for each layer. 3) Integrated all the unoptimized layers to build the full ALexNet model for synthesis. 4) Integrated all the optimized layers to build an optimized working model of AlexNet architecture.

We have built the HLS model of all layers by starting with Lab1 and Lab2 implementations as skeletons and making modification or additions as per the requirement of each layer.

Following are some of the techniques we used to optimize the HLS implementaions:
1) Array Partitioning
2) Tiling
3) Unrolling and Pipelining
4) Data quantization
5) Loop Reordering
6) Efficient data reading and writing - optimizing DRAM access
7) Dataflow - memory streaming

We will elaborate on some of the most important and helpful techniques in the next section.

IV. OPTIMIZATION TECHNIQUES

A. Tiling

As mentioned earlier in the Problem Description, we initially focused on bringing the BRAM utilization down because of the limited resources on board. The use of "Tiling" in order to bring a small block of input data and layer parameters helped in achieving this. Using trial and error method, we found out optimal tile sizes that give the best combination of utilization and latency for each of the layers separately. The dimensions used can be seen in the "conv.h" file in the HLS implementation codes.

B. Data Quantization

We used different lengths of fixed type data in the range of fixed_type $\langle 16, x \rangle$ and fixed_type $\langle 32, x \rangle$ and observed the accuracy, speedup and utilization associated with each of them. This was done to decide a length of data which gives the best combination of the three parameters. We have used the data type of $\langle 16, 4 \rangle$ in the HLS implementation of individual layers as well as integrated implementation.

C. Dataflow

We have used the "dataflow" pragma available in Vitis HLS to streamline the reading, compution and writing operations. We also made a manual ping-pong buffer for one convolution layer. But we observed that the Utilisation estimates as well as latency wer better after using the dataflow pragma. That is why, in the final implementation we have used the pragma.

V. RESULTS

We used virtexuplus xqvu7p-flra2104-1-I FPGA board for synthesizing AlexNet (all layers integrated). We has to size up to a larger FPGA from the Pynq-Z2 or Ultra96 because of the high resource requirement of all 14 layers.

Figure 5 shows the final resource utilization after integrating all unoptimized layer, giving us a final latency of 15.6 seconds. Figure 6 shows the final resource utilization after integrating all optimized layers, givig a final latency of 1.877 seconds i.e. speedup of 8.3x from unoptimized version.

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+---------------------+---------+------+---------+--------+-----+
|         Name        | BRAM_18K|  DSP |    FF   |   LUT  | URAM|
+---------------------+---------+------+---------+--------+-----+
|DSP                  |        -|     -|        -|       -|    -|
|Expression           |        -|     -|        0|       2|    -|
|FIFO                 |        -|     -|        -|       -|    -|
|Instance             |     3532|    67|    14855|   28759|    0|
|Memory               |    55118|     -|        0|       0|    0|
|Multiplexer          |        -|     -|        -|    1533|    -|
|Register             |        -|     -|      116|       -|    -|
+---------------------+---------+------+---------+--------+-----+
|Total                |    58650|    67|    14971|   30294|    0|
+---------------------+---------+------+---------+--------+-----+
|Available SLR        |     1440|  2280|   788160|  394080|  320|
+---------------------+---------+------+---------+--------+-----+
|Utilization SLR (%)  |     4072|     2|        1|       7|    0|
+---------------------+---------+------+---------+--------+-----+
|Available            |     2880|  4560|  1576320|  788160|  640|
+---------------------+---------+------+---------+--------+-----+
|Utilization (%)      |     2036|     1|       ~0|       3|    0|
+---------------------+---------+------+---------+--------+-----+
```

Fig. 5. Unoptimized integration of layers

The utilization reports of individual optimized and unoptimized layers is shown in the table at the end of the report.

VI. DEPLOYMENT

We successfully deployed CONV1 layer of AlexNet on the Pynq-Z2 board with full functional correctness. Figure 4

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+---------------------+---------+------+---------+--------+-----+
|         Name        | BRAM_18K|  DSP |    FF   |   LUT  | URAM|
+---------------------+---------+------+---------+--------+-----+
|DSP                  |        -|     -|        -|       -|    -|
|Expression           |        -|     -|        0|     284|    -|
|FIFO                 |        -|     -|        -|       -|    -|
|Instance             |      666|   212|    27459|   50412|    0|
|Memory               |     1194|     -|        0|       0|    0|
|Multiplexer          |        -|     -|        -|    4391|    -|
|Register             |        -|     -|      203|       -|    -|
+---------------------+---------+------+---------+--------+-----+
|Total                |     1860|   212|    27662|   55087|    0|
+---------------------+---------+------+---------+--------+-----+
|Available SLR        |     1440|  2280|   788160|  394080|  320|
+---------------------+---------+------+---------+--------+-----+
|Utilization SLR (%)  |      129|     9|        3|      13|    0|
+---------------------+---------+------+---------+--------+-----+
|Available            |     2880|  4560|  1576320|  788160|  640|
+---------------------+---------+------+---------+--------+-----+
|Utilization (%)      |       64|     4|        1|       6|    0|
+---------------------+---------+------+---------+--------+-----+
```

Fig. 6. Optimized integration of layers

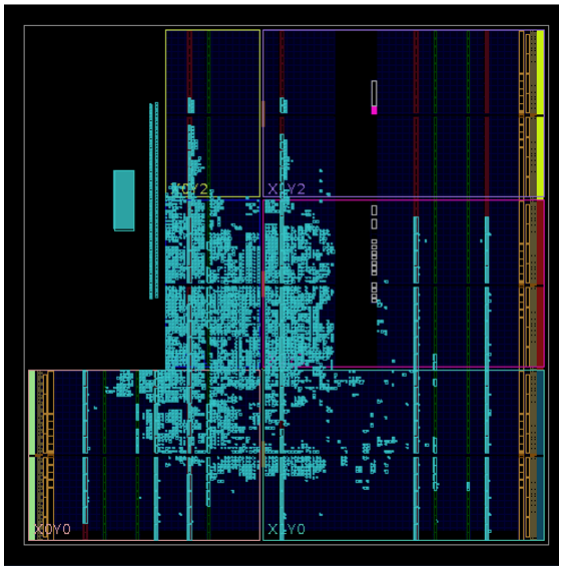

Fig. 7. CONV1 design

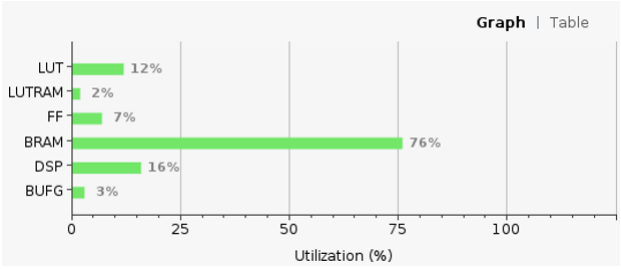

Fig. 8. CONV1 utilization report

shows the design of CONV1 in Vivado post bitsteam generation. Figure 5 shows the resource utilization of CONV1 on Pynq-Z2 board.

For this, we performed the RTL/ C Cosim using the Vitis GUI. The Cosim passed! After this, we imported the RTL design of CONV1 to Vivado and generated the bitstream for the same.

Uploaded the .bit, .hwh and .bin files (input, weights, bias and golden output) to pynq cluster on Jupyter notebook and checked the outputs returned by FPGA to the golden outputs, which matched. The observed on-board execution time was 4.3 seconds, compared to 1.3 seconds recorded in the Vitis HLS report.

VII. DSE & LEARNING

A. Data Quantization

For our HLS implementation, we tried different lengths in bits for the fixed type data used. Various combinations of bits from 16 bits to 32 bits were tried and the latency, accuracy and resource utilization for each was observed. Finally, we tried to find which gives us the best combination of all three, i.e. latency, accuracy and utilization. We found $\langle 16, 4 \rangle$ fixed type data to give the best results for project. All individual layers as well as the integration of all layers were done using $\langle 16, 4 \rangle$ datatype throughout.

B. Resource limitation on board

During our implementation, we had to let go of some optimization techniques that were mentioned in our reference paper, presented by us in the class [3]. Like, in the paper we presented, the output of each layer was being stored on board using BRAMs and fed as input to the next layer. But due to limited BRAM resources available to us, we had to store the outout of each layer back into the DRAM and then load it back to the BRAM for the next layer. This walk all the way to the DRAM after each layer added to our final latency. Had we had more BRAMs available to us, we could have followed the same approach of storing intermediate layer outputs on board itself.

C. Math functions in HLS

Two of our layers have math library functions. First, LRN has a power function and second, FC3 was using the exp function for it's sigmoid activation function. We were successfully able to synthesize these two layers using the "hls_math.h" library and get the unoptimized latency. Eventually, for the optimized implementation, we linearized our power function in LRN layer and replaced the sigmoid activation function with ReLU in the FC3 layer under professor's guidance. This helped us bring down the resource utilization by a lot, and moreover also made the integration of optimized layer simple.

REFERENCES

[1] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems 25 (2012)

[2] K. Jarrett, K. Kavukcuoglu, M. A. Ranzato, and Y. LeCun. What is the best multi-stage architecture for object recognition? In International Conference on Computer Vision, pages 2146–2153. IEEE, 2009

[3] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou and Lingli Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural networks," 2016 26th International Conference on Field Programmable Logic and Applications (FPL), 2016, pp. 1-9, doi: 10.1109/FPL.2016.7577308.

Layers			BRAM	DSP	FF	LUT	LAT (ms)
		Available	432	360	141120	70560	
Conv1 (3x227x227)	UNOPT	TOTAL	414	16	4012	7001	8441
		UTIL	95%	4%	2%	9%	
	OPT	TOTAL	159	21	5136	10308	1244 (6.78x)
		UTIL	36%	5%	3%	14%	
LRN1 (96x55x55)	UNOPT	TOTAL	536	3	3291	3899	8.713
		UTIL	124%	$\sim$ 0%	2%	5%	
	OPT	TOTAL	4	0	1479	2245	2.9 (3x)
		UTIL	$\sim$ 0%	$\sim$ 0%	1%	3%	
Maxpool1 (96x55x55)	UNOPT	TOTAL	2205	5	1522	3553	4.304
		UTIL	510%	1%	1%	5%	
	OPT	TOTAL	194	5	13406	19389	2.968 (1.45x)
		UTIL	44%	1%	9%	27%	
Conv2 (96x27x27)	UNOPT	TOTAL	828	12	3835	6805	8969
		UTIL	191%	3%	2%	9%	
	OPT	TOTAL	27	11	10637	10855	47.66 (188.2x)
		UTIL	6%	3%	7%	15%	
LRN2 (256x27x27)	UNOPT	TOTAL	175	4	1224	2136	11.213
		UTIL	40%	1%	$\sim$ 0%	3%	
	OPT	TOTAL	2	0	930	1514	1.86 (6x)
		UTIL	$\sim$ 0%	0%	$\sim$ 0%	2%	
Maxpool2 (256x27x27)	UNOPT	TOTAL	1428	3	1651	3346	2.732
		UTIL	330%	0%	1%	4%	
	OPT	TOTAL	2	1	39345	57841	1.885 (1.45x)
		UTIL	$\sim$ 0%	0%	27%	81%	
Conv3 (256x13x13)	UNOPT	TOTAL	973	12	3499	7174	3001
		UTIL	225%	3%	2%	10%	
	OPT	TOTAL	47	20	11217	17091	11.57 (259.4x)
		UTIL	10%	5%	7%	24%	
Conv4 (384x13x13)	UNOPT	TOTAL	260	5	2727	4863	252
		UTIL	60%	1%	1%	6%	
	OPT	TOTAL	223	200	9146	15378	5.252 (47.98x)
		UTIL	45%	1%	1%	6%	
Conv5 (384x13x13)	UNOPT	TOTAL	197	5	2722	4834	168
		UTIL	45%	1%	1%	6%	
	OPT	TOTAL	172	69	5634	12260	4.362 (38.51x)
		UTIL	39%	19%	3%	17%	
Maxpool5 (256x13x13)	UNOPT	TOTAL	349	1	1353	3113	0.617
		UTIL	80%	0%	0%	4%	
	OPT	TOTAL	2	0	9740	15265	0.441 (1.4x)
		UTIL	$\sim$ 0%	0%	6%	21%	
FLAT (9216)	UNOPT	TOTAL	24	0	1017	2167	0.277
		UTIL	5%	$\sim$ 0%	$\sim$ 0%	3%	
	OPT	TOTAL	4	0	1126	3230	0.092 (3x)
		UTIL	$\sim$ 0%	$\sim$ 0%	$\sim$ 0%	$\sim$ 0%	
FC1 (9216x4096)	UNOPT	TOTAL	34581	2	2761	4151	1133
		UTIL	8004%	$\sim$ 0%	1%	5%	
	OPT	TOTAL	44	32	3925	5441	379 (3x)
		UTIL	10%	8%	2%	7%	
FC2 (4096x4096)	UNOPT	TOTAL	29720	6	2124	3374	336
		UTIL	6879%	1%	1%	4%	
	OPT	TOTAL	27	16	3892	5255	167 (2x)
		UTIL	6%	4%	2%	7%	
FC3 (4096x10)	UNOPT	TOTAL	51	18	4319	5257	1.23
		UTIL	11%	5%	3%	5%	
	OPT	TOTAL	8	16	4376	5104	0.459 (2.7x)
		UTIL	1%	4%	3%	7%	