Efficient Implementation of Convolutional Neural Network on FPGA in OpenCL

Shreya Deodhar & Chinmay Joshi,
Department of Electrical and Computer Engineering
Clemson University, Clemson, SC - 29634, USA

Abstract- **In this report we present a hardware architecture to implement a feed-forward convolutional neural network. We present a performance comparison between a software implementation and two different hardware approaches on a Cyclone V FPGA that shows a speed-up in custom hardware implementations.**

We present results in terms of execution time comparisons, kernel time comparisons and resource utilizations for the Cyclone V FPGA. We have observed a maximum speed-up in the parallel approach, which was found to be 1.77. We found that parallel approach is more suitable for the problem, and can be further optimized by better memory management.

I. INTRODUCTION

The Deep learning is emerging as a promising tool for solving virtually any complex big data problem. Deep Learning have dramatically outperformed state-of-the-art in computer vision, speech recognition and so on. A Convolutional Neural Network (CNN) is made up of neurons that contain weights and biases that are trainable [11]. Each neuron in a CNN obtains some inputs and executes a dot product.

Typical complex implementation of neural net contains many convolutional layers, each followed by a pooling layer and fully connected sigmoidal-softmax layers at the last. The Convolution layer is the fundamental unit of a Convolutional Network, and its output volume can be taken to be a plot of neurons arranged in a three dimensional volume i.e., the neurons are organized in three dimensions namely the width, height, depth. Now if all neurons in a single depth slice are using the same weight vector, then the forward pass of the Convolutional Layer can in each depth slice be computed as a convolution of the neuron's weights with the input volume [11]. Thus the layer's parameters consist of a set of learnable filters. Each of these filters is small and spatial in two dimensions and height. It extends through the full depth of the input volume. The filtering process will be designed in order to slide the filter, across the input, and compute the dot product between the entries of the filter and the input to get the output of the layer. Thus during the forward pass, we will slide i.e. convolve each filter across the width and height of the input, thus producing a two dimensional movement pattern of that filter. By stacking this activation maps for all filters along the depth, the complete output volume can be obtained. Each entry in the output volume will therefore also be interpreted as an output of a neuron that looks at only a small region in the input and shares parameters with neurons in the same activation map.

CNN's are thus popular in image classification problem. Fig [1] shows CNN architecture to implement MNIST Database Classification [3]. They are using 28 X 28 images, 20 feature maps, pooling by factor of 4 , 100 sigmoid and 10 softmax neurons.

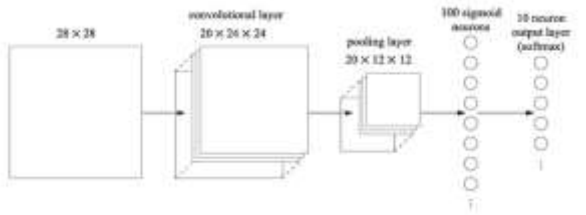

Fig 1. Typical CNN Architecture [3]

II. BACKGROUND AND LITERATURE SURVEY:

Convolutional Neural Networks:

Hubel and Wiesel's work [4] in 1968 showed that mammal's (cat's) visual cortex contains complex structure of cells and tendency of cells to be grouped according to symmetry of responses to movement.
Convolutional neural network is bio-inspired structure to implement Feed-forward neural network. Convolutional neural networks work best on data which has strong local correlation such as Images.
Convolutional neural networks use different filters to extract different feature maps from the input data. These feature mapped are then further processed by subsequent convolutional or sigmoids layers. Convolutional approach saves number of weights compared to fully connected layers.

RELU Activation function:

RELU (Rectified Linear Unit) is a type of activation function used in artificial neural networks.

$$f(x) = \max(0, x)$$

Sigmoid Activation function:

Sigmoid/tanh functions are commonly used as activation function in neural network, as they give good non-linearity as well their derivative is computationally efficient to calculate.

Pooling:

Pooling is operation in which we reduce the size of intermediate data (often output of convolutional layer) with minimum loss of intelligence. Max-Pooling and Average pooling are some types of pooling. Sigmoid layers are fully connected. Hence, having pooling layer before sigmoid layer reduces computation significantly, without losing much intelligence.

Softmax function:

Softmax or normalized exponential is a gradient log classifier which is useful to classify sparsely spread values in consistent manner. Softmax function is routinely used as a last layer of neural networks for classification purpose.

$$O_j = \frac{e_j}{\sum_i e_i}$$

Zang, Chen et al., implemented Convolutional neural network on Xilinx VC707 FPGA board with 61.62 GFLPOS efficiency [1]. They used roofline model for designing FPGA kernels to balance computational and memory access optimizations.

Ovtcharov, Kalin, et al. presented Microsoft research white paper on Microsoft's catapult project in which deep convolutional neural networks are accelerated using Stratix V D5 FPGA's.[6]

III. IMPLEMENTATION

We implemented the feedforward Convolutional neural network having a fixed filter of size of 3 X 3. The max-pooling factor is 4. We are using 10 sigmoid and 10 softmax functions. We could not fit more sigmoid units on the fabric. Fig [2] shows our architecture.

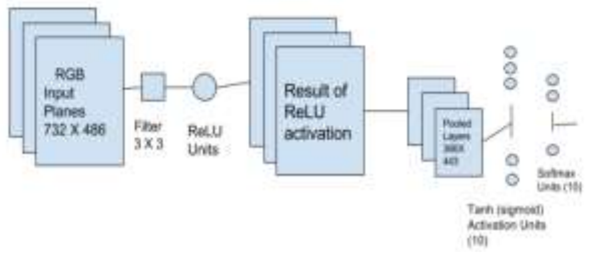
RGB Input Planes 732 X 486
Filter 3 X 3
ReLU Units
Result of ReLU activation
Pooled Layers 366X 443
Tanh (sigmoid) Activation Units (10)
Softmax Units (10)

Fig 2. Our Architecture

Parallel Approach:

The problem stated above seems to be embarrassingly parallel in its first stage i.e. 2D Convolution, and so we designed the first approach to be parallel. Thus the implementation, is a simple way to parallelize 2D image convolution is by writing the OpenCL code in such a way that W X H threads are launched. Each thread thus launched has the task of grabbing Filter Size X Filter Size X 2 pixels per thread from the global memory of the FPGA. Of these, Filter Size X Filter Size are taken from the input image to be convolved, whereas the second Filter Size X Filter Size are from the filter being used for convolution. Because every pixel obtained requires 3 channels (red, green and blue), and taking into account that the convolution operation will specifies different filtering conditions for every channel, the design needs 6 accesses for each pixel. This brings the number of global memory reads to (6 x Filter Size X Filter Size).

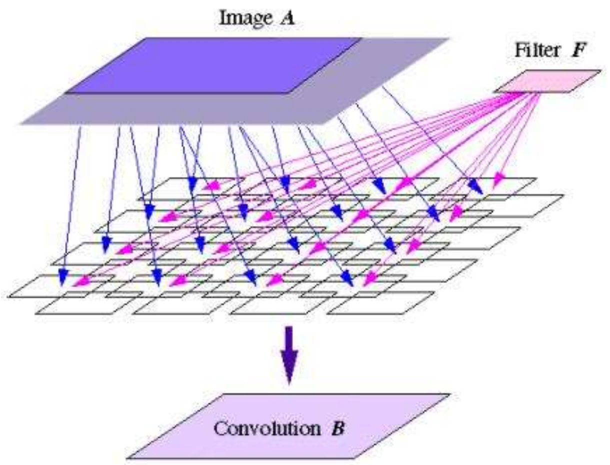

Fig 3. Parallel Computation of convolution (Image Source: [8])

Thus for an image of dimensions H X W, there are H X W threads working parallely to compute each index of the output array of the convolution function. Each of these process threads then send their output to a channel to be sent to the second kernel which computes rest of the operations of the other layers of the convolutional network. These consist of max-pooling, normalization and rectification.

We then noticed that the convolution filter does not change during the complete execution and thus we stored it in the constant memory rather than storing it in global memory as before. The constant memory for the FPGA being faster than the global memory gave us even better speedup in both the parallel as well as the pipelined approaches as illustrated in the Fig [5] and Fig [6]. Being in the constant memory, the filter is precached before launching the kernel. This memory is limited for an FPGA, but is sufficient for a large filter size.

No of RGB pixels	Timing for Kernel1	Timing for Kernel2	Overall timing	overhead	With __Constant buffer
10000	148.709	148.586	162.73	14.144	142.273
40000	594.121	593.995	648.686	54.691	568.396
90000	1337.739	1337.617	1459.418	121.801	1279.97
160000	2375.903	2375.777	2591.477	215.7	2272.95
355752	5282.168	5282.056	5748.273	466.217	5053.35
1000000	14848.201	14848.085	16154.26	1306.18	14204.5

Table 1. Parallel Implementation Timings

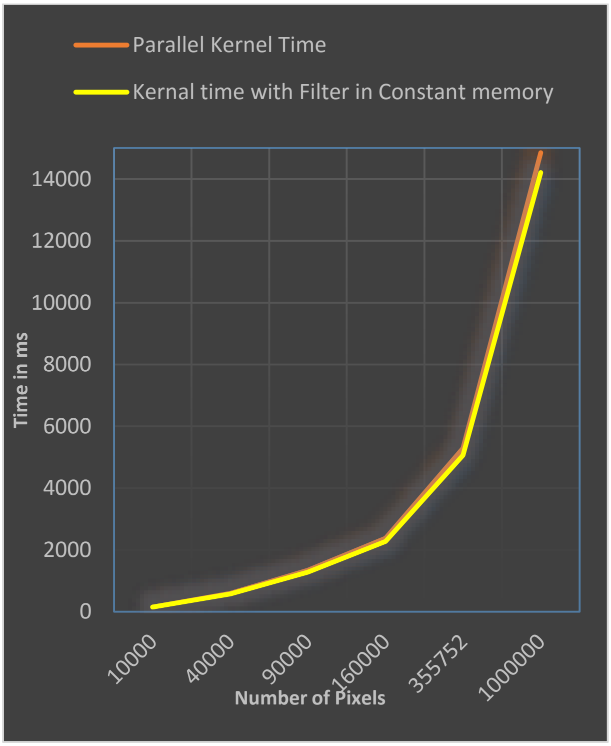

Fig 4 Pipeline Implementation: __Constant buffer Comparison

After making the filter a constant memory component also we will still need (6 x Filter Size X Filter Size) memory accesses for the computation of calculating a convolution. The only difference is that 3 X Filter Size X Filter Size accesses are to faster constant memory thus reducing the overall execution time of the implementation.

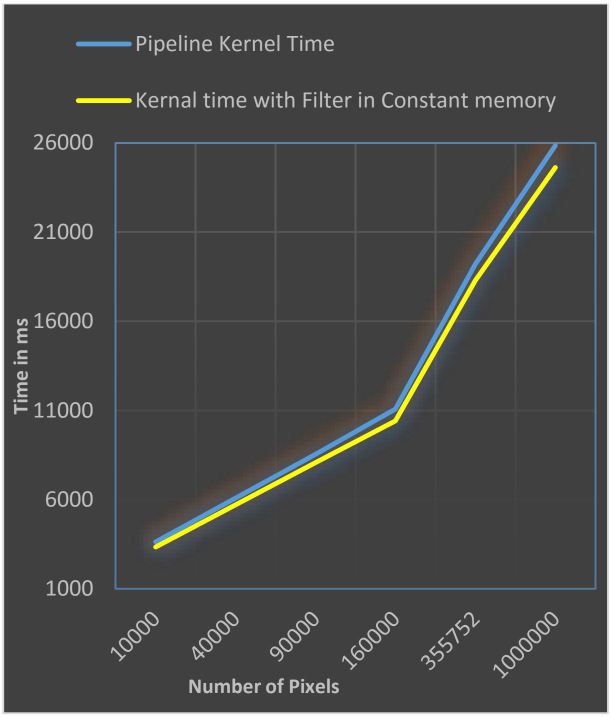

Fig 5 Pipeline Implementation: __Constant buffer Comparison

Pipelined Implementation:

As we described above, the parallel approach for the Convolution stage in the neural network is memory access intensive and this increases the kernel-time due to the large memory access latency. In order to study the effect of this latency, we decided to implement a completely pipelined approach.
For this we referred to the design example of a Sobel filter by Altera on an FPGA. The example stated contains of a single work-item kernel that has been designed to implement a Sobel operator in order to detect edges in an input RGB image and give a monochrome image as output.
We have also implemented a kernel code which has a sliding-window line buffer. We used this approach to reduce the memory accesses to the global memory of the device.

Thus the feed forward neural network works as shown in the diagram below. The data for only the required part of the image is fetched during a single iteration. On every cycle the previous unwanted data is discarded and a new data packet is fetched after shifting all the previous data by the same amount as that of the discarded data in order to get an effect of the sliding window.

Every stage of the Convolutional Neural Network works on the principle of processing the data available currently in a pipeline.

This implies that the max pooling starts when convolution data for the Width * 3 + Filter Size has been obtained. After this an output is obtained on every alternate cycle.

No of RGB pixels	Timing for kernel	Overall Timing	Communication host processing overhead	Constant
10000	3628.285	3630.004	1.719	3338.438
40000	6081.773	6086	4.227	5703.869
90000	8537.274	8548	10.726	8070.877
160000	11052	11070	18	10420.855
355752	19165.508	19204	38.492	18290.031
1000000	25755.874	25864	108.126	24622.612

Table 2. Pipelined Implementation Timings

IV. OPTIMIZATIONS:

1. Math functions Optimization:

Tanh is an asymptote function defined as

$$\tanh(x) = \frac{x}{1+\frac{x^2}{3+\frac{x^2}{5+\dots}}}$$

Hence we can optimize tanh for fewer iterations to give us less precise but fast results.
Similarly, for exponential function,

$$exp(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} \dots\dots$$

We can optimize exponential function also for less precise but acceptable results with fast computation. If we try to use math.h functions tanh() and exp() for floats, aocl creates considerably large hardware.[7]

2. --fp-relax --fpc options in AOC compiler:

AOC compiler can do optimizations to create pipelined and efficient hardware implementation for floating point operations. These optimizations are can cause small differences (often negligible for non-critical floating point math) and not part of IEEE standard 754-2008. AOC, by default, complies with IEEE 754-2008 and other OpenCL standards.
--fp-relax enables balancing of floating point operation tree, thereby pipelining and reducing the hardware required.

Rounding operations in floating point math are expensive. --fp-relax tree balancing produces more rounding operations. Multiple rounding operations lead to less accurate results and less optimized hardware. AOC --fpc option reduces rounding operations and maintains precision by additional mantissa bits. It also supports implementation of fused multiply-accumulate (FMAC) structures.
With --fp-relax and --fpc option we got 15% more efficient hardware utilization.

V. RESULTS

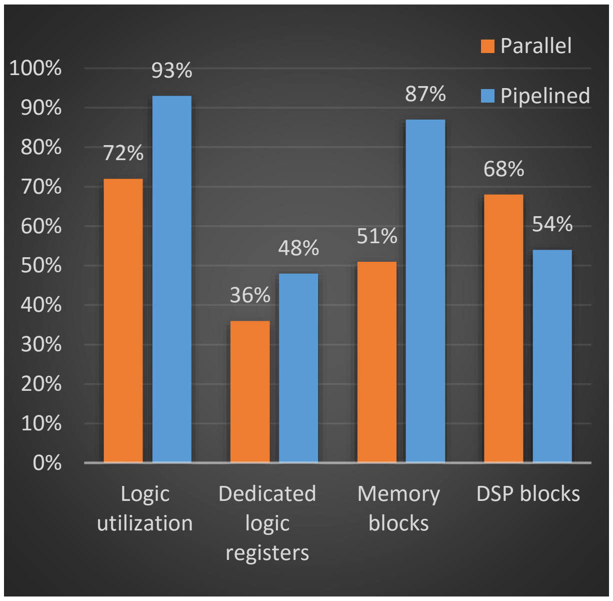

Fig 6. Comparison of Logic Utilization

The figure [6] shows a comparison of resource utilization for the two FPGA implementations presented in the report. It can be seen that logic utilization and logic registers for pipelined approach is greater than the parallel one. This result has been obtained because the logic for shifting the data in the sliding-window line buffer is additionally added to the design. It can also be seen that the memory blocks for parallel approach is less than that of the pipelined approach this is because we are allocating additional memory to the shift registers for the sliding window effect. The DSP block utilization for the parallel implementation is found to be more than the pipelined approach. This is the result of parallel hardware generated for the floating point computation in the convolution kernel.

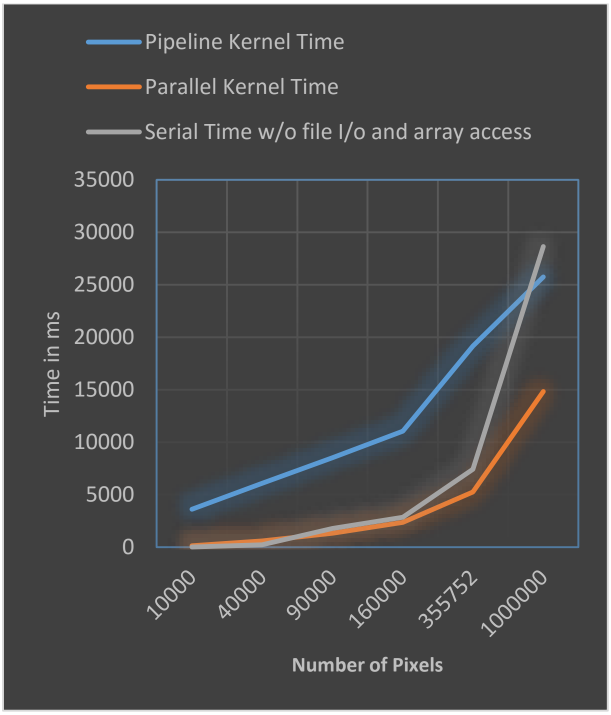

Fig 7. Comparison of Timings

The figure [7] gives a comparison of timing diagrams for the different approaches implemented and described before. The serial implementation has been implemented in C programming language and the timings have been recorded while running it on an Intel CPU having a clock rate of 1107.917 MHz. The clock rate for the parallel implementation in OpenCL was found to be 107.09MHz. Also the clock rate for the pipelined implementation was found to be 105.03MHz.
It can be observed that the parallel version gave us a speed-up right from the smaller data sizes of 300 X 300 i.e.- 90000 pixels increasing with the increase in data size. For the pipelined approach however we observed the speed-up at a very large data size of 1000 X 1000 pixels.

The figure [8] shows a comparison between the addition of communication and host code times for the two approaches implemented on the Cyclone V FPGA. It can be seen that this time is more for the parallel implementation. Also this time is increasing exponentially for the parallel implementation along with the increase in the input data size.

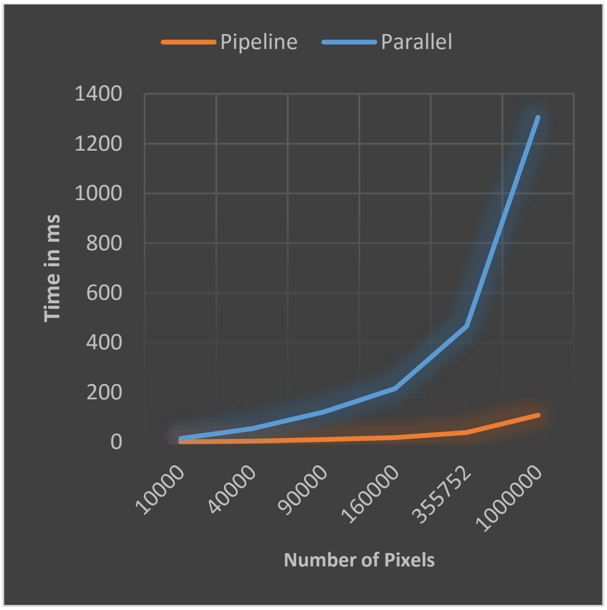

Fig 8. Overhead time comparison

Profiling Results:

We profiled our parallel implementation using AOC --profile option. This option compiles performance counters inside the kernels and creates a file on host side with this information.
We are getting 78.13% efficiency when reading global buffer, while just 33.33% efficiency when reading another global buffer(Weights) from host side. When we investigated the reason, we actually found bug in our code which causes the second buffer be accessed in non-sequential order.

We are also getting very low occupancy of channels. The reason for this may be that we are using 3 different channels. If we can restructure our input data, we can use only one channel for data transfer. Our first kernel, conv_kernel is multi work-item kernel while second kernel, sigm_kernel is one work-item. The conv_kernel sends data to sigm_kernel using 3 channels. We are getting 99.94% stall on writing part of channels, because multiple threads are contending for channels. We can avoid this by using on-chip memory and avoiding use of channels.

VI. FUTURE WORK:

1. Optimizations:

We can optimize our kernels more from inputs we got from profiling. We can optimize our input data format and data type. We are currently using 3 floats for one pixel. Whereas, we can easily pack one pixel in 24 bits. We can use fixed point representation instead of floating point, and implement tanh and exp for fixed point representation. Transfer of weights is also major bottleneck. We can eliminate this need by random weight generation on kernel for a training application.

2. Implementing Back-propagation:

With a larger device such as Arria 10, we can implement Back-propagation on the fabric. The main motivation behind our project was to accelerate learning process.

3. Real time processing of images:

We can take images in real time with USB cam attached to SoC board. We can interface the webcam using V4L2 API on Embedded Linux running on ARM core. We can also provide real time video output using one of the port attached to SoC. This can turn into very interesting project where we identify image-frame on the fly. This project can be integrated in autonomous vehicle detection, where we can get very reliable real-time detection.

4. Neural network with pruning and other optimizations:

We can investigate how we can integrate optimizations such as pruning, weight decay on FPGA implementations. We may have to use Reconfiguration feature in order to implement such dynamic learning techniques.

VII. CONCLUSION

From this report, we can conclude that our parallel implementation is better in terms of logic utilization and speed-up. This would mean that large latency due to multiple memory access is not really a big problem. However, when considering multiple images being processed at a time, we might still face problems due to the projected delays. This needs to be explored more. With profiling result for parallel implementation, we found that we need to max-out our channel use and avoid stalling. Rather than having only one copy of the second kernel access the data from multiple copies of the first kernel, if we can accept the values into multiple copies of second kernel, we might reduce stalling. For this, we need to parallelize our second kernel, which requires significantly higher logic area. Other approach would be to use a buffer in second kernel to smooth channel performance to some degree. In pipelined version, we can unroll many loops inside if we use larger device. Pipelined approach would actually be faster than parallel one, if we could introduce this parallelism to it on sufficiently larger fabric.

We witnessed how control branching disables pipelining. We used some techniques to avoid some if-else blocks. These solutions can be counter-intuitive and look inefficient but they do provide good results on hardware. We learned many aspects of parallel computing and FPGA design with this project. Also this project has helped us get a good understanding of, how we can fit a practical neural network on a larger FPGA board.

ACKNOWLEDGMENT

We would like to thank Dr. Melissa C. Smith, Nilim Sarma, Brad Green and Sufeng Niu for all the help and advice provided during the course of this project.

REFERENCES

[1] Zhang, Chen, et al. "Optimizing fpga-based accelerator design for deep convolutional neural networks." Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2015.

[2] http://www.deeplearning.net

[3] Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015

[4] Hubel, David H., and Torsten N. Wiesel. "Receptive fields and functional architecture of monkey striate cortex." The Journal of physiology 195.1 (1968): 215-243.

[5]https://www.altera.com/support/support-resources/design-examples/design-software/opencl/sobel-filter.html

[6] Ovtcharov, Kalin, et al. "Accelerating deep convolutional neural networks using specialized hardware." Microsoft Research Whitepaper 2 (2015).

[7] http://math.stackexchange.com/questions/107292/rapid-approximation-of-tanhx,

[8] http://www.johngustafson.net/pubs/pub11/Benchmark6-pack.htm

[9] “Altera SDK for OpenCL Programming Guide." (2015)

[10] “Altera SDK for OpenCL Best Practices Guide” (2015)

[11] http://cs231n.github.io/convolutional-networks/