Convolutional Nets on FPGA
Final Report

Ricson Cheng
ricsonc

Sree Vishant Prabhakaran
sreevisp

May 12, 2017

1 Summary

We implemented inference with convolutional neural networks in SystemVerilog on the FPGA. We improved the naive baseline strategy, and compare the performance of both our initial and final FPGA implementations with an implementation of convolutional neural networks on CPU and on GPU.

2 Background

2.1 System Overview

Our convolutional neural network is a system which takes images as inputs, and outputs a number from 1 to 10 in order to classify the image into one of ten categories. The network consists of three types of components: convolution layers, pooling layers, and activation layers.

Each of these components takes as input a feature map and produces a new feature map. A feature map is simply is a three dimensional array (height x width x depth), where the set of values along the depth axis is called a single "feature". For example, the input RGB image is a feature map with each feature being the RGB value at a single pixel.

A pooling layer takes as input a feature map and outputs a feature map with half the height and half the width. In order to do this, it combines 2x2 blocks of adjacent features into one feature by applying a "pooling function". We used the max pooling function, which simply takes the maximum of each group of four input features.

An activation layer takes as input a feature map and applies the same activation function to every item in the map, producing an identically sized feature map. For our activation

layer, we used a variant of the "rectifier linear unit", which is defined as $f(x) = \max(0, x)$.

A convolution layer takes as input a feature map and also takes as input a set of filters, which is a four dimensional array (filter size x filter size x filter depth x number of filters). Then, it convolves the feature map with the filter, producing a new feature map. To be precise, if I is the input feature map with depth d and F is the filter with size $2r + 1$, then the output I' is defined below.

$$I'[i][j][k] = \sum_{\Delta_i=-r}^{r} \sum_{\Delta_j=-r}^{r} \sum_{l=1}^{d} F[\Delta_i][\Delta_j][l] * I[i + \Delta_i][j + \Delta_j][l]$$

Finally, our network consists of a sequence of these layers. The first layer has the input feature map being the input image, which is 32x32x3. The last layer has its output feature map being the of the class of the image, which is 10x1x1. The network operates by having each layer execute its task when the previous layer has completed.

2.2 Parallelism

All three components of the network can be parallelized. The activation layer is particularly easy to parallelize because all elements in the output depends on only one input element, so they can all be independently computed. For the same reason, the pooling layer can also be pretty easily parallelized, since each output element is dependent on four input elements, which no other output depends on.

However, the most benefit from parallelization comes from the convolution layers, which are the most computationally expensive. In addition, the convolution layers are more challenging to parallelize because of the potential for performance gains from optimizing for data locality.

3 Approach

3.1 Details

We implemented our project using SystemVerilog in Vivado. We targeted the Zedboard FPGA. Due to time constraints, we were not able to run the code on the FPGA, but we were still able to analyze the performance using the simulator provided in Vivado (details in the Results section). We wrote all of our code from scratch.

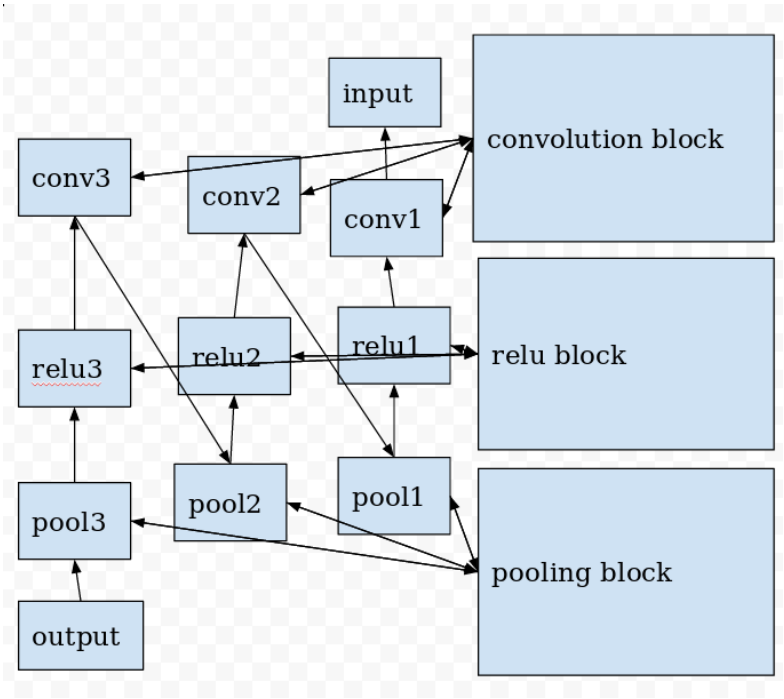


Figure 1: Our final implementation of a convolutional network on FPGA is depicted here. The arrows are used to denote data-dependencies. In contrast to the naive implementation, depicted further down, in this implementation, each "layer" of the network is a very lightweight controller which interfaces with the convolution, relu, and pooling blocks. The convolution, relu, and pooling blocks are relatively large, but they can be efficiently reused.

3.2 Final Approach

We describe our final approach to the problem here (our original approach is detailed later). Our final approach makes use of convolution, pooling, and activation "blocks". Each block consists of a large number of units, a single unit being a module which can compute the activation function, or a module which can perform a dot product between two small vectors. These blocks allow us to do 1024 pooling operations at once, or 1024 activations at the same time. In a way, doing computations with these blocks is similar to doing SIMD computation, just on a larger scale. We implemented this blocking architecture in our SystemVerilog code.

Then, in order to implement a convolution layer, we instantiate a convolution "controller", which is given control of the convolution block. The controller handles the movement of data in and out of the block, and uses it to convolve the feature map it is given.

Careful design of the convolution controller was needed to achieve the best performance.

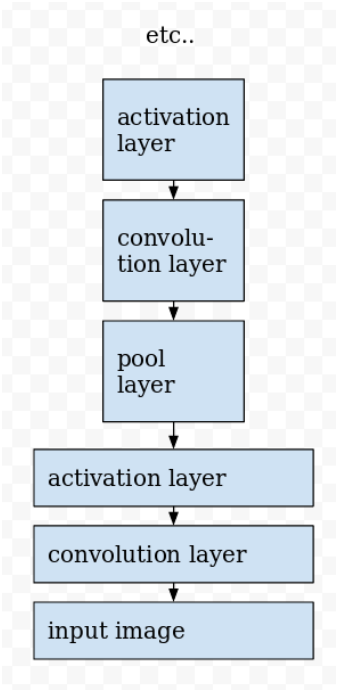


Figure 2: The naive implementation of a convolutional network on FPGA is depicted here. The arrows are used to denote data-dependencies. Because of the completely linear structure of this implementation, logic elements are not efficiently reused.

We pipelined the usage of the convolution blocks. A naive implementation requires two clock cycles to run the convolution block – one cycle to place the inputs, and another cycle to extract the computed outputs from the block. However, we managed to pipeline the operation by feeding inputs of iteration $t + 1$ into the block while simultaneously reading outputs from time t, doubling the throughput of the module.

Another detail the convolution controller had to handle was convolving with multiple filters. In the convolution equation in section 2.1 we used l for the index of the filter. However, simply executing the summation in that equation would be inefficient, because the filter index changes in the inner loop, resulting in poor data locality. Therefore, we convolved the entire feature map with the first filter, then convolved the entire feature map again with the second filter, and so on. Our method of reordering the loops in the convolution decreases the number of clock cycles spent loading filters into memory.

The pooling and the activation controllers were very similar to the convolution controller, but simpler. However, we used the same pipelining trick in order to speed up computation there as well.

3.3 Other attempts

Our original approach was to use a combinatorial implementation of pooling, activation, and convolution. By combinatorial, we mean that the output was a state-less function of the input wires. This approach has the advantage of low latency, since the time needed to compute the output is simply the time needed for electricity to propagate from the input wires to the output wires.

However, this approach also has the disadvantage that it makes poor use of the hardware. For example, the first activation layer in our network has an feature map size of 32x32x8, so we instantiated on hardware 8192 activation modules, each of which was used only once. The lack of state prevented us from reusing an activation module to compute the activation of more than one value.

4 Results

4.1 Overview

We measured both performance in terms of time and power consumption of several baselines and our implementations. We measured time needed to send a single image through a convolutional neural network. We compared the outputs of all implementations to ensure correctness. Details of the network are described in the Appendix. The four implementations tested are listed below.

1. CPU implementation (Intel Pentium G3258)

2. GPU implementation (Geforce GTX 1060)
3. FPGA implementation A (our first try)
4. FPGA implementation B (out final results)

We note that the synthesized FPGA-A implementation used more 800% logic elements than were available on the Zedboard, and therefore is physically impossible to synthesize. Nonetheless, it serves as a baseline.

4.2 Methodology

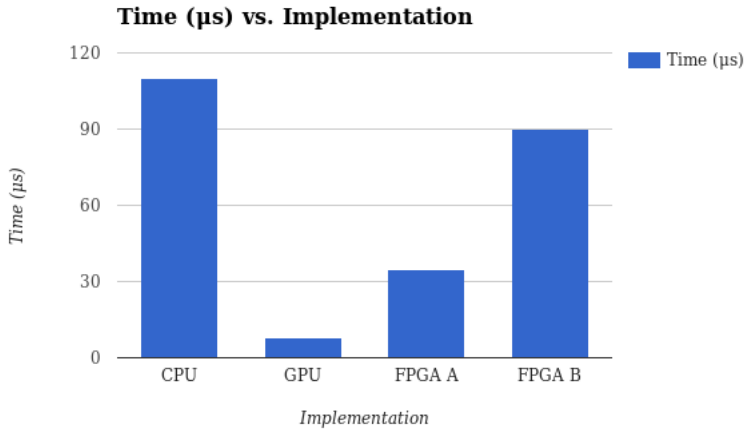
For our CPU and GPU baselines, we measured performance of the network implemented using Keras, with Tensorflow as the backend. These are highly optimized implementations for convolutional networks, so we would expect them to do very well. The CPU implementation takes advantage of multiple cores (2) and also SIMD instructions. We averaged cost over one million images to obtain accurate results. To measure power consumption for GPU, we used the command-line tool "nvidia-smi". Measuring power consumption for CPU was much more difficult, and we settled for a estimate of 40W based off of estimated idle power usage and processor TDP. We used the Vivado software the time needed on the FPGA. We encountered difficulty when attempting to measure energy consumption on the Zedboard FPGA. Therefore, we use a liberal estimate of 3W as the maximum power draw from a Zedboard. Even using this figure, we find that FPGA computation is more efficient, which is not surprising.

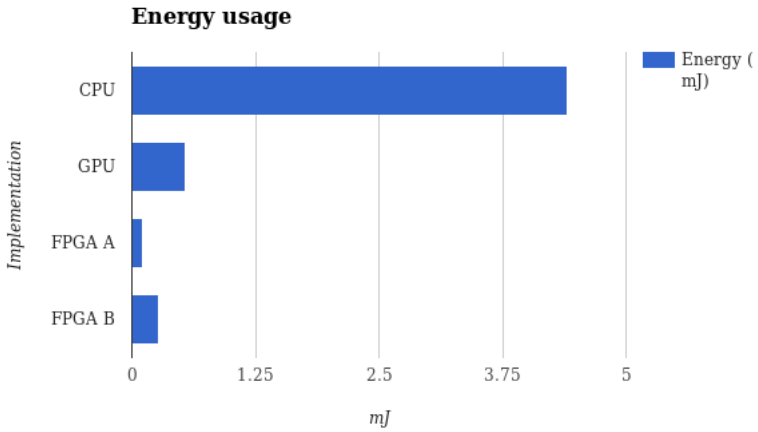
4.3 Measurements

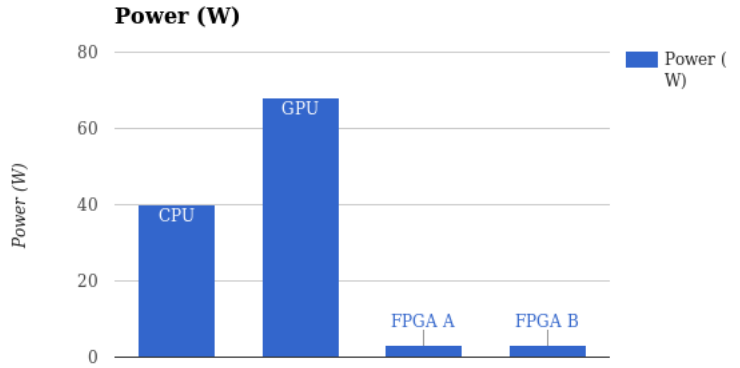
All values are given per-image.

Implementation	Time (μs)	Energy (mJ)	Power (W)
CPU	110	4.4	40
GPU	8	0.54	68
FPGA-A	35	0.11	3
FPGA-B	90	0.2790	3

The above table in graph form:


Time (µs) vs. Implementation
120
90
60
30
0
Time (µs)
CPU
GPU
FPGA A
FPGA B
Implementation
Time (µs)


Energy usage
CPU
GPU
FPGA A
FPGA B
Implementation
0
1.25
2.5
3.75
5
mJ
Energy (mJ)


Power (W)
80
60
40
20
0
Power (W)
CPU
GPU
FPGA A
FPGA B
Power (W)

4.4 Discussion

In terms of raw performance GPU is the fastest piece of hardware. However, both our FPGA implementations use about half as much energy per image than the GPU. Although our second FPGA implementation was not better than our first FPGA implementation in terms of speed, this is because we allowed our first implementation to use more logic elements than are available.

4.4.1 Scaling

We reported results for images of size 32x32x3. However, we can still effectively reason about the performance at different image sizes. Our first implementation would not work at all at a larger image size, due to the fact that it is fully combinatorial.

On the other hand, we would expect our second implementation to scale very well with input image size, up until the image gets too large to fit into the FPGA LUT memory.

4.4.2 Limitations

The limiting factor of our implementation comes from the sequential structure of the layers of a convolutional network. Even though we have a convolution block, an activation/relu block, and a pooling block, only one of the three blocks is used at any one time, since the data must pass through the layers one at a time.

This claim can be made more rigorous by the following mathematical analysis. In our network, each feature map must have HWD convolutions applied to it, HWD activations applied, and $HWD/4$ pooling operations applied. Note that a convolution takes exactly the same time as a pooling or an activation operation, because we implemented this combinatorially. We adjusted the block sizes such that three of these steps takes the same amount of time, therefore it is mathematically true that only one out of three logic elements is being utilized at any time.

Furthermore, another limiting factor was the time it took to move bits around the FPGA. According to the Vivado synthesizer, along the critical path, about 2/3rds of the runtime was due to data routing and 1/3rd was from logic computations.

4.4.3 Target

We chose as our target machine an FPGA platform because of its large amount of parallelism and energy efficiency. In terms of just throughput, it is difficult to compete with a GPU. However, we believe that trying to use an FPGA is worth doing because energy efficiency is important, and because it is valuable to explore how parallelization of a certain task can be very different between different pieces of hardware.

Note that due to time constraints, we did not end up running our code on the actual FPGA. As mentioned above, all measurements were carried out by simulation in Vivado.

5 References

Thomas, Donald. "Logic Design and Verification Using SystemVerilog". CreateSpace Independent Publishing Platform, 2014.
Abadi, Martín et al. "Tensorflow: Large-Scale Machine Learning on Heterogenous Systems". 2015.
Krizhevsky, Alex. "Learning Multiple Layers of Features from Tiny Images". 2009.

6 Appendix

6.1 Quantization

Neural networks are typically implemented with 32-bit floating point integers. Due to the relative difficulty and inefficiency of floating point arithmetic on FPGA, we opted to quantize the weights of our neural network and do computations using 16-bit integers.

We multiplied all weights and biases of the trained network by 1024, and then truncated the values. To compensate for this, immediately following every convolution in the FPGA-implementation, we divide by 1024 (which can be done efficiently) in order to balance out the shift. We find empirically that there is little to no error induced by the quantization process we used.

6.2 Network and Dataset

We used the CIFAR-10 dataset, which is comprised of 50,000 color images, each of which is 32 by 32 pixels in size. The dataset has 10 categories.

The layers of our network are listed below in order

- convolution (3x3) with 4 filters
- relu activation
- pooling layer
- convolution (3x3) with 8 filters
- relu activation
- pooling layer
- convolution (3x3) with 16 filters

- relu activation
- pooling layer
- convolution (1x1) with 8 filters
- relu activation
- convolution (3x3) with 16 filters
- relu activation
- pooling layer
- convolution (1x1) with 8 filters
- relu activation
- convolution (3x3) with 16 filters
- relu activation
- pooling layer
- convolution (1x1) with 8 filters
- relu activation
- convolution (1x1) with 10 filters
- softmax activation

Note: The softmax layer takes 10 inputs and converts them into values which can be interpreted as probabilities. We did not implement the softmax layer on FPGA, because is it not needed to perform inference.

6.3 Work by each student

Equal work was performed by both project members.