Project Proposal

Kyle Daruwalla
Department of Electrical and Computer Engineering
University of Wisconsin – Madison
daruwalla@wisc.edu

Akhil Sundararajun
Department of Electrical and Computer Engineering
University of Wisconsin – Madison
asundararaja@wisc.edu

1 Introduction

Machine learning systems tackle problems ranging from content filtering and recommender systems to object recognition and text classification. In recent years, advances in deep learning have led to finding better classifiers for these problems. In deep learning, feature extraction is performed automatically by using many layers of neural networks; each layer involves passing inputs of the previous layer through a nonlinear activation function. Compositions of successive layers can learn nonlinear decision boundaries, which has contributed to successes in image classification and speech recognition [1]. Training of deep neural networks is performed using the stochastic gradient descent (SGD) algorithm, but the inherently serial nature of SGD has led to exploration of potential speedups through parallelized hardware. In this project, we seek to investigate how the computational cost per iteration of training a deep neural network depends on the hardware architecture being used.

1.1 Field-Programmable Gate Arrays

Even though learning algorithms are inherently serial, speedup might be possible by using specialized hardware to reduce the cost per iteration.

Field-programmable gate arrays (FPGAs) are reconfigurable hardware chips. An FPGA is comprised of *slices*, which are the fundamental hardware unit from which any designed hardware is constructed. Each slice is comprised of *look-up tables* (LUTs) and *flip-flops* (FFs). When reporting the resource consumption of a particular design, it is common to report the metric in terms of slices or LUTs+FFs.

Hardware on an FPGA is designed using a *hardware description language* (HDL). The most common HDL is Verilog. While Verilog shares some syntax with C, it should not be confused for a sequential programming language. HDLs allow a designer to spatially describe the hardware.

FPGAs are commonly used for real-time control, because the design freedom they offer allows for lean, efficient controller design. Furthermore, designs are not hampered by hardware limitations, because the designer can create any hardware he desires. As the boundary between control theory and optimization has blurred, FPGAs have become suitable hardware platforms for machine learning algorithms such as neural networks [2] [3]. Similarly, FPGAs are an attractive option to make object-recognition algorithms real-time [4].

While previous work has largely focused on deployment of neural networks on FPGAs, this project will focus on the training phase. Specifically, can FPGAs be utilized to build efficient parallel hardware to speedup the lengthy training process for convolutional neural networks?

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

2 Design Methodology

In order to perform an effective comparison between software implementations of CNNs and hardware implementations, we use TensorFlow, MATLAB, and Xilinx FPGAs. TensorFlow is used to train software implementations of a given CNN structure for CIFAR-10. These CNN implementations were trained and tested on a traditional CPU. A hardware implementation of the CNN has been created for Xilinx FPGAs in Vivado. Simulations have been performed to verify its functionality and collect timing characteristics, while an equivalent MATLAB model verifies its convergence.

2.1 TensorFlow Baseline

A software implementation of a CNN was created in TensorFlow to serve as a baseline for comparison. In order for the network to fit on a single FPGA chip, we chose to only implement a simple, two-layer CNN. The structure for such a network is shown Fig. 1. The TensorFlow implementations were

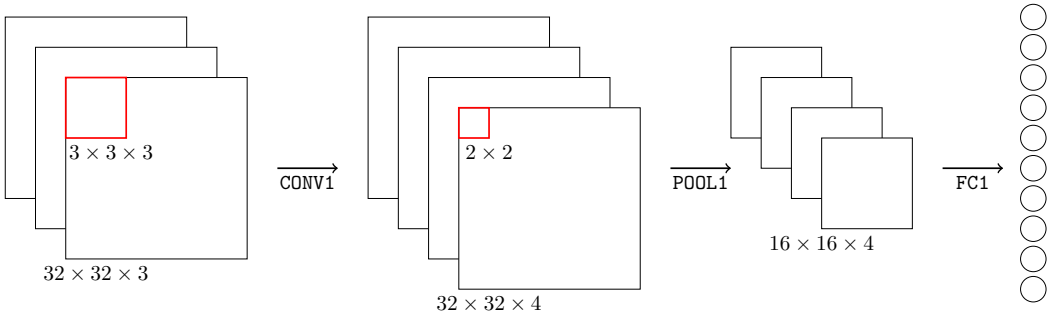

Figure 1: Our CNN architecture structure

trained using 12,800,000 samples using mini-batch stochastic gradient descent. We trained the model using Amazon EC2 spot instances with `c2x.large` (8-Core Xeon) and `c8x.large` (32-Core Xeon) processors.

2.2 Convolutional Neural Networks on FPGAs

Our FPGA design for convolutional neural networks is an attempt to decrese the cost per iteration of training a CNN. While it does use some techniques that may change the convergence of the neural network, algorithmically, it is serially equivalent to software implementations for training a CNN. This is important, because we are not trying to create a new parallelized algorithm such as HOGWILD!, we are only trying to apply traditional hardware speed-up techniques to CNNs.

Currently, most of the computation cost of a CNN is concentrated in the convolutional layers. Furthermore, attempts at hardware parallelization have been hampered by the cost of memory accesses. In order to tackle these issues, we designed a fundamental *convolutional filter unit*. Each filter unit represents a single convolutional filter in a CNN, and each layer of a CNN can contain several convolutional filters. We wanted to run these filters in parallel, so that we only pay the cost of sliding the filter over the entire input image space once per layer. Thus, we needed to keep the resource cost of each filter low enough that multiple filter units can be instantiated in the hardware. To do this, we use 16-bit fixed point (14 fractional bits, 1 sign bit) hardware with stochastic quantization throughout the FPGA implementation [5]. Stochastic rounding or quantization is described by Eq. 1 where ϵ is machine epsilon (i.e. machine precision) for a given fixed point system.

$$\text{Round}(x) = \begin{cases} \lfloor x \rfloor & \text{with probability } 1 - \frac{x - \lfloor x \rfloor}{\epsilon} \\ \lfloor x \rfloor + \epsilon & \text{with probability } \frac{x - \lfloor x \rfloor}{\epsilon} \end{cases} \tag{1}$$

Gupta et al. describe how to implement such a rounding mechanism in hardware in the presentations accompanying their paper [5]. Fig. 2 rounds a higher precision number by adding to the lower (discarded) fractional bits, as is common in round-to-nearest quantization schemes. However, stochastic rounding adds a pseudorandom number to the lower bits instead of pre-determined number. The pseudorandom number is generated using a linear feedback shift register (LFSR) as detailed in Fig.

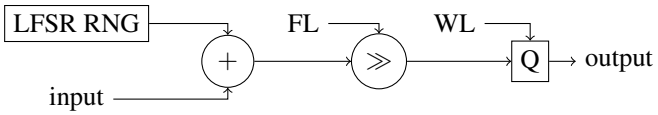

Figure 2: Stochastic rounding quantizer (FL is fractional length; WL is word length; Q selects the WL LSBs of its input)

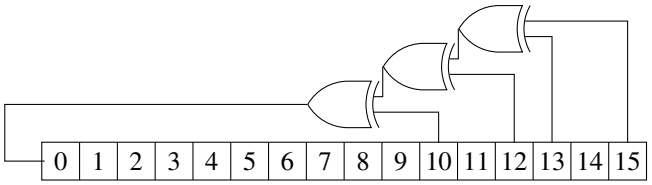

Figure 3: 16-bit linear feedback shift register

3. In addition to reducing the hardware resources required to perform arithmetic operations, using 16-bit fixed point numbers significantly reduces the computation time of an adder or multiplier, and it reduces the memory footprint required to store numbers. These benefits will be important when we discuss the design of each filter unit.

In addition to using fixed point numbers to speed up computation, we exploit spatial locality and operational level parallelism when design the filter units. Each unit contains its own register file to store the weights and biases related to it. This way, the weight and bias vectors do not need to be fetched or stored in memory, and they do not need to be passed around. As a result, the filter unit is capable of not only computing the feed-forward pass for each training sample, but also propagating error back through the network and using the error to compute gradients and update its own weights. Lastly, the major operation performed by a filter unit is a vector dot product. This involves a series of multiplications and additions. The multiplications are performed in parallel, and the additions use a binary tree-like structure to perform as many additions in parallel as possible. For an $n \times 1$ vector, our addition structure takes $\log(n) \cdot d$ delay (where d is the delay for a single addition), while a naive implementation takes $(n-1) \cdot d$ time. Fig. 4 illustrates a block level diagram of a filter unit (though the diagram includes some control signals, it does not include all of them to save space).

Though Fig. 4 illustrates a 5×1 vector of inputs and weights, our implementation uses a 3×3 patch which results in 9×1 vector of inputs. During a feed-forward pass, the filter unit is fed with a patch of inputs. An input contains a 3D tensor, but each patch is only 2D. Control signals are used to inform the filter unit which slice of inputs along the third dimension it is operating on. Accordingly, it fetches the correct inputs performs the dot product. An accumulator register stores the output of successive dot products as patches along the third dimension are supplied to the filter unit. After traversing across the depth dimension, the filter unit quantizes the result of the dot product and adds the bias. This result is passed through σ and σ'. The σ output is a single pixel of output from the layer, and the σ' is stored for later use.

During the feed-backward pass, the filter unit first propagates the error by convolving the rotated weight kernel with the error from the previous layer. It is provided with the previously stored σ' values through the activation derivative input. The output of this step is stored pixel by pixel as the error for the convolutional layer.

The next stage is the gradient update stage. The filter unit is provided with the appropriate input map and error map to compute the gradient for the appropriate subset of the weight kernel. The error patches are also accumulated using a binary tree adder structure and an accumulator register. This result is the gradient update for the bias. After performing the convolution and accumulation operations, the gradients are used to update the bias and weight values.

On a higher level, we perform convolution by first accumulating across the depth dimension, then sliding across horizontal and vertical directions. We perform max pooling by sliding a pooling filter across the input horizontal and vertical directions. Across the depth dimension, max pooling is done in parallel. The fully-connected layer is done on 16×1 patches of vectors at a time. But the fully-connected patch operation is performed in parallel for all 10 output elements.

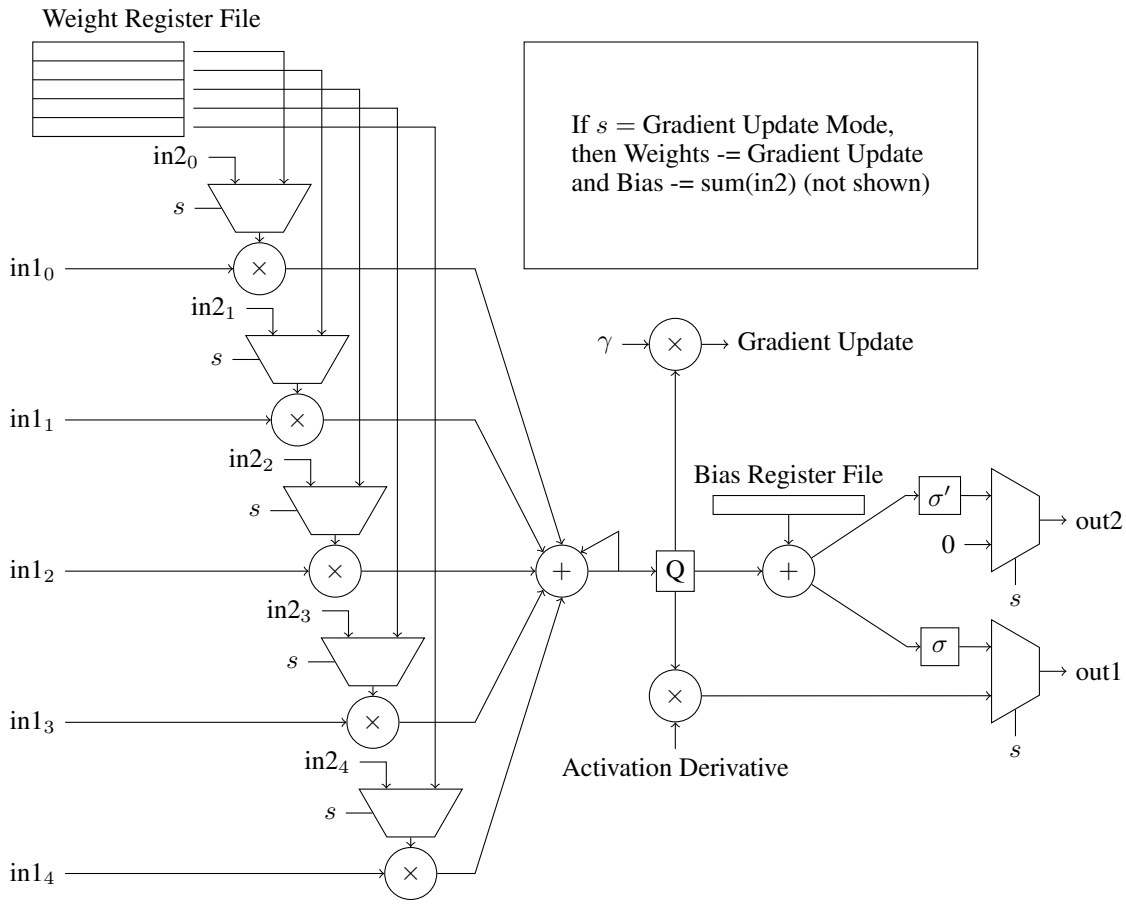

Figure 4: A convolutional layer filter unit

2.3 MATLAB Verification

While our hardware implementation is serially equivalent to a traditional software implementation for a CNN, it does use limited precision. We implemented a limited precision CNN model of the hardware in MATLAB to verify its convergence.

This was achieved by using a custom quantization function to limit the precision of the numbers used by MATLAB. Using a custom function allowed us to have tight control of the rounding scheme, and it is faster than using the object-oriented fixed-point model in MATLAB.

Similar to the TensorFlow and hardware, the MATLAB simulation is also trained using mini-batch stochastic gradient descent with a batch size of 128 (same as TensorFlow). The purpose of this simulation was to emulate training the hardware on the same dataset as TensorFlow verify the hardware's convergence. It was not intended to be used for a timing comparison.

3 Results

We perform a total training time comparison between TensorFlow implementations and the FPGA hardware. Additionally, we perform a loss convergence comparison between the simulated hardware in MATLAB and the TensorFlow implementation.

3.1 Total Runtime Results

After designing the hardware, we simulated the Verilog in Vivado to find the total number of clock cycles to train one batch of mini-batch SGD. We the multiply this value by the operating clock period

of the FPGA to determine to absolute time measurement to train a single batch for various clock frequencies. Lastly, we multiply that value by the total number of batches to find the total training time for the given FPGA implementation.

Fig. 5 displays a comparison of the total training time in minutes between the TensorFlow implementations and the FPGA implementation at varying clock frequencies. As can be seen in the plot, the FPGA needs to be run at 50 MHz minimum to notice any speed up.

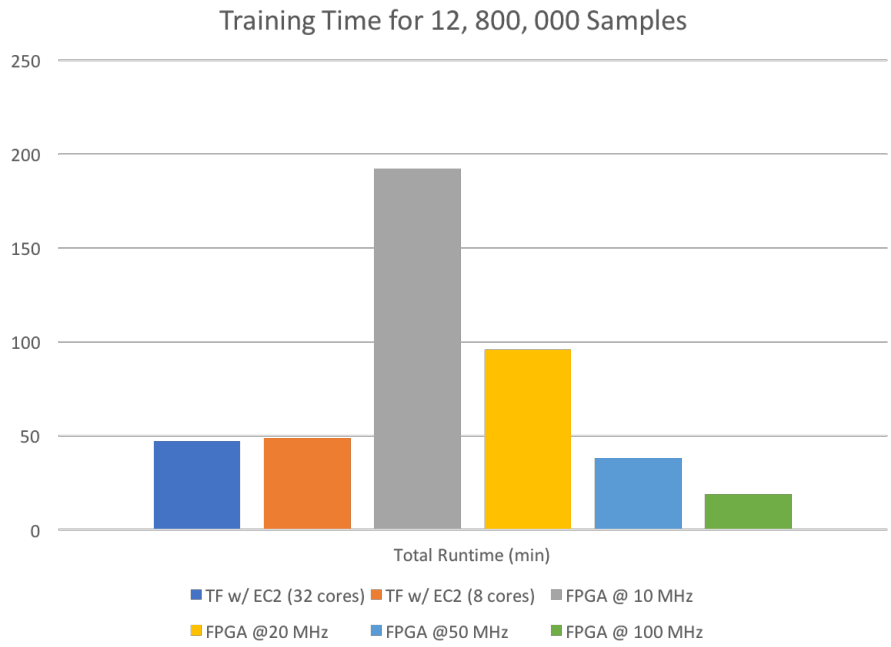

Figure 5: A comparison of training time for 12,800,000 samples

Notice that the TensorFlow implementation notices little to no speed-up going from 8 to 32 cores. This illustrates the lack of potential in current traditional strategies for achieving speed-up by increasing the number of cores or machines. The FPGA implementation, however, scales linearly with frequency. The largest deterrent of running the FPGAs at 100 MHz or higher, is the lack of on-chip resources. Specifically, attempting to fit an entire CNN on a single FPGA introduces too much global routing overhead, which leads to a lower maximum clock frequency. We suspect that by partitioning the neural network by layers onto multiple FPGAs, the hardware will be able to run at the higher frequencies required to achieve speed-up.

3.2 Loss Convergence Results

We trained the MATLAB simulation of the hardware using the same dataset as the TensorFlow implementation to verify its convergence. Fig. 6 illustrates the convergence of the loss versus number of samples processed. Clearly, the loss converges, but it coverges to a worse loss than the full precision TensorFlow implementation. The purpose of this analysis was primarily to verify that the hardware is capable of producing a trained CNN. Initially, we expected our loss convergence plots to look similar to Gupta et al. [5]. However, our plot shows a noticeable difference in the loss value that each implementation converges to. We expect that this is due to the difference in size between our network and the networks implemented by Gupta et al. Since our CNN contains siginificantly less layers and learned parameters, the effect of losing a bit of precision is amplified. In a larger network, the layers can work together to mitigate the effects of losing precision. We believe this discrepancy in the loss convergence plots lends creedence to the need to explore the effects of limited precision training more in depth.

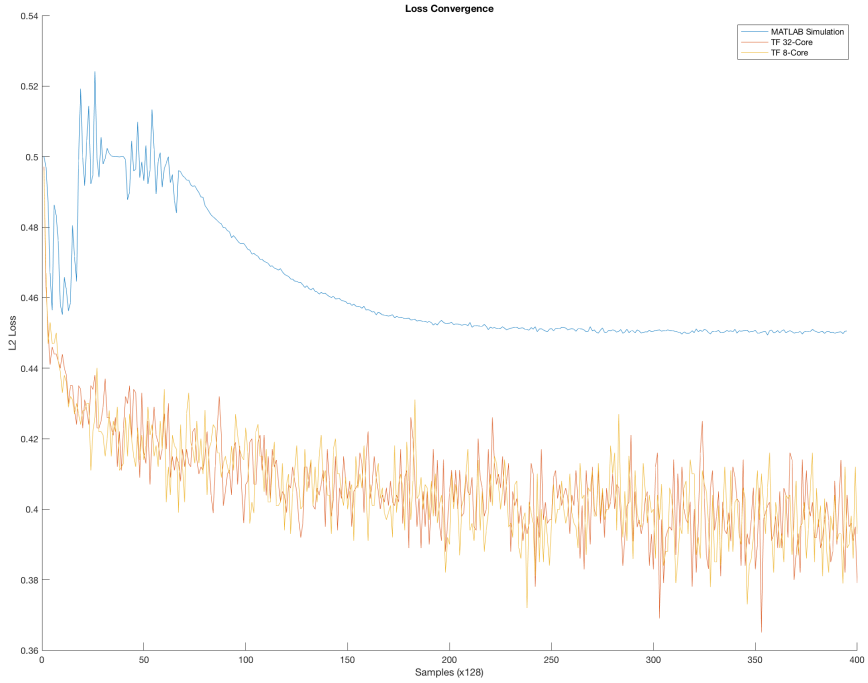

Figure 6: The loss versus number of samples processed for the MATLAB and TensorFlow implementations

4 Conclusion

Our results show that FPGAs are a viable option for speeding up the training phase for CNNs. However, we also note that these implementations are constrained by the resources available, and thus, FPGA implementations are only useful if the network can be partitioned. In particular, training a small to medium sized networks on a single CPU or GPU is likely a better option than an FPGA implementation. However, as the input data and the network depth scales up, the benefits of CPU implementations fail to deliver speed-up. Particularly, at the scale of data centers like Microsoft [6], an FPGA implementation offers the necessary speed-up while still being more power efficient than an equivalent GPU offering.

Furthermore, our loss convergence results illustrate the need to explore limited precision training more deeply. Currently, neural network designers add controlled noise to the data or model (via dropout) in order to increase the model robustness. We would like to explore using quantization error to introduce this noise. The properties of quantization error can be controlled and bounded via varying rounding schemes. Moreover, the benefits of quantization and limited precision are two-fold – first, controlled noise can make the model more robust if inserted intelligently; second, the reduced bit length leads to faster hardware and more resources on-chip to create parallel execution units.

In summary, this project illustrates a promising direction for further exploration. In the future, we would like to implement partitioning networks across several FPGAs to increase the clock frequency and number of parallel execution units, as well as understand the effects of quantization and limited precision on the robustness of CNNs. We conclude that FPGAs are a suitable candidate for speed-up when the networks and datasets are sufficiently large, and when aggregate power consumption is costly.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," Nature, 2015.

[2] J. Wang, Y. Chen, J. Xie, B. Chen, and Z. Zhou, "FPGA based neural network PID controller for line-scan camera in sensorless environment," Fourth International Conference on Natural Computation, 2008.

[3] J. Skodzik, V. Altmann, B. Wagner, P. Danielis, and D. Timmermann, "A highly integrable FPGA-based runtime-configurable multilayer perceptron," IEEE 27th International Conference on Advanced Information Networking and Applications, 2013.

[4] B. Ahn, "Real-time video object recognition using convolutional neural network," International Joint Conference on Neural Networks, 2015.

[5] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *CoRR*, vol. abs/1502.02551, 2015.

[6] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," Microsoft Research, 2015.

[7] F. Niu, B. Recht, C. Ré, and S. J. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent." arXiv:1106.5730v2, 2011.

[8] M. Abadi, P. Barham, J. Chen, and Z. Chen, "Tensorflow: A system for large-scale machine learning," CoRR, 2016.