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

The first experiment will be to study training performance of the ImageNet dataset on Amazon EC2, which will serve as a baseline for a traditional single-machine CPU setting. We propose to use TensorFlow, an open-source, python-based machine learning package released by Google in November 2015 [5]. In addition, using packages in TensorFlow, we will study GPU acceleration on Amazon EC2. Due to TensorFlow's distributed execution capability, we also propose to train CNNs using the HOGWILD! algorithm. Since SGD is inherently serial, our FPGA implementation targets lowering cost per iteration. This can be compared to HOGWILD!, which is a parallelized approach to achieving speedup. Due to the variety of structural and parameter design choices typical in building a CNN, we also propose to examine several CNN architectures in combination with the different hardware platforms considered in this project. Different activation functions including sign, ReLU, and sigmoid also will be explored.

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 [6]. 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 [6]. Fig. 1 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. 2. In addition to reducing the hardware resources required to perform arithmetic operations, using

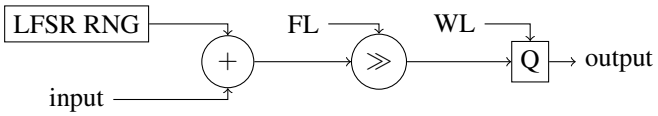

Figure 1: Stochastic rounding quantizer (FL is fractional length; WL is word length; Q selects the WL LSBs of its input)

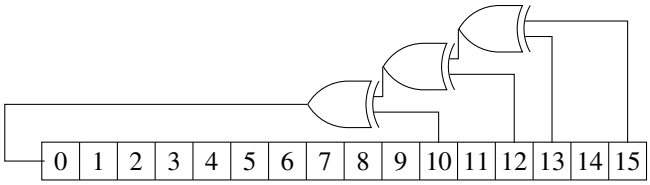

Figure 2: 16-bit linear feedback shift register

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. 3 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).

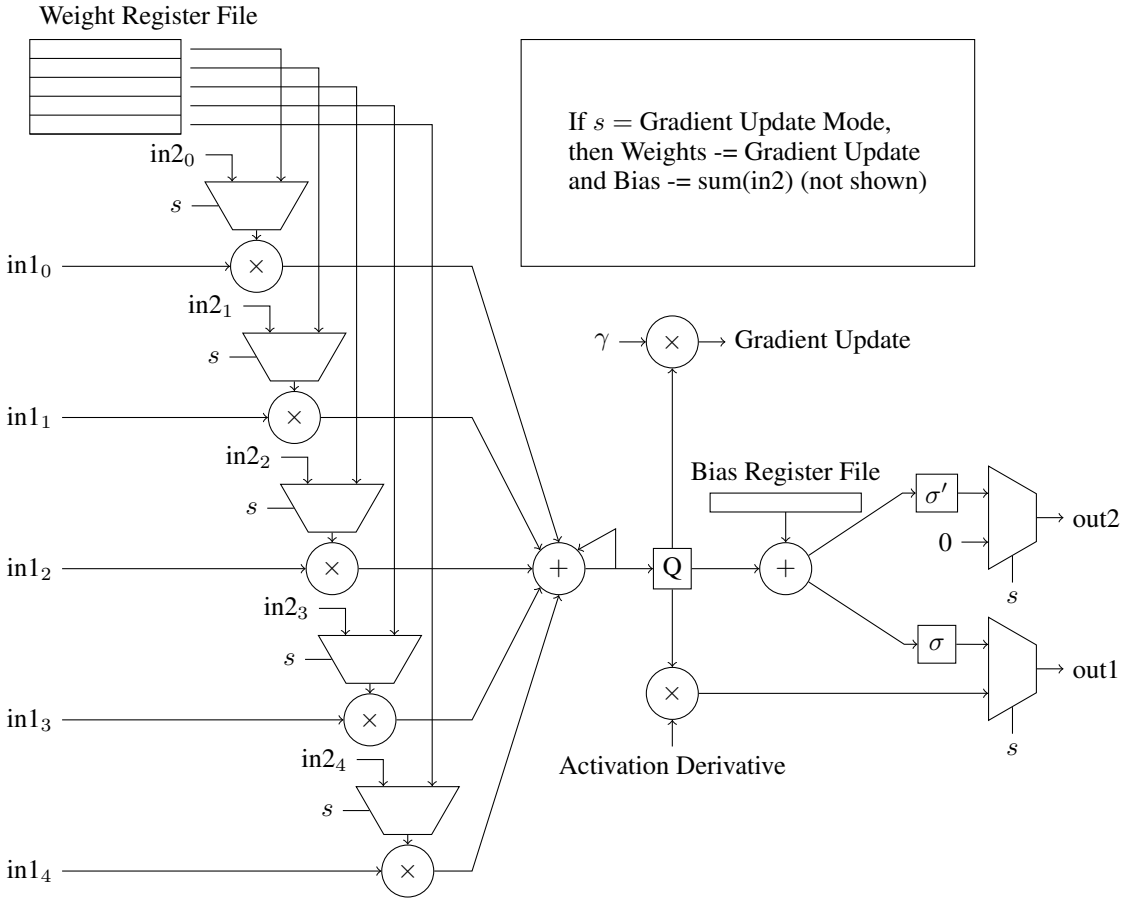

Figure 3: A convolutional layer filter unit

Though Fig. 3 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.

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.

3 Results

After creating an FPGA implementation of a neural network, timing characteristics for a particular CNN can be created. For example, the proposed unit-neuron will have some physical path delay, τ. Using this known constant and the structure of our CNN, we would like to define a design space. This will allow a designer targeting neural networks on FPGAs to know the time per iteration as a function of τ, the number of layers, and the CNN structure. This will provide some theoretical closed form for the computational complexity on an FPGA.

Additionally, after collecting empirical results, we would like to do perform the following analysis:

1. Comparing generalization error between the CPU, GPU, HOGWILD!, and FPGA implementations.
2. Comparing convergance rates (in terms of seconds) for SGD with backprop between the CPU, GPU, HOGWILD!, and FPGA implementations.
3. Provide a metric of when FPGAs might provide a larger speedup than HOGWILD!

4 Conclusion

Insert conclusion.

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

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