REDUCED FLOATING POINT APPROXIMATE COMPUTING IN MACHINE LEARNING: EFFECTS ON SPEED, MEMORY, POWER, AND ACCURACY

Dillon Pulliam [*1] **Mehrnoosh Raoufi** [*2]

https://github.com/JDPulliam/HwArcML_FinalProj/tree/MNIST
https://github.com/JDPulliam/HwArcML_FinalProj/tree/CIFAR10
https://github.com/JDPulliam/HwArcML_FinalProj/tree/Vivado_HLS

ABSTRACT

In this project our team explores reduced floating point approximate computing in machine learning by analyzing a variety of different floating-point number schemes in regard to deep neural networks. This analysis is done on a number of different hardware platforms: CPUs, GPUs, and FPGAs. While running these tests and doing this analysis, results are categorized from the different number schemes on corresponding hardware. The main factors analyzed are speed, memory usage, power consumption, and model accuracy.

Analysis begins at a high-level by implementing various architectures in Python and running them on a CPU. After this is completed, code is added to allow training and inference on GPU's where results are again categorized using the same models and number schemes. Finally, some MLP architectures are selected to implement on a FPGA using Vivado High-Level Synthesis (HLS) and are used to classify results before experimenting with a custom Verilog implementation.

1 INTRODUCTION

Neural network break-throughs are happening at an extraordinary pace with each typically being correlated to a more complicated model larger in size. Moreover, more complex networks require higher computational power to train and consequently have larger memory footprints. Thus, improving speed, power, and memory consumption has become critical. Reduced floating point number schemes are an approximate computing technique that can offer an alternative to address these issues on numerous platforms. In this project, we explore how this technique can benefit machine learning in terms of speed, power, and memory while minimizing accuracy loss. In this section, we discuss three aspects that can potentially be benefitted.

1.1 Speed

One benefit to reduced floating point approximate computing is speedup both in regard to training and inference time. The motivation behind speeding up training is that if it can be done faster more models can be trained, and thus more time spent tuning hyper-parameters for optimization. In terms of inference, one motivating factor is that some applications must maintain a specific minimum latency to be useful. For instance, in self-driving cars decisions must be made quickly or else run the risk of causing potential accidents or safety hazards. Therefore, we hope that by implementing reduced floating point number schemes we can achieve speedup in training and inference, particularly on applications where latency is critical.

1.2 Power

Another potential benefit from reduced floating point number schemes is a reduction in power consumption. From a cost point of view, this may be considered the most important factor. As the complexity of a neural network grows, so does power consumption which is correlated with the number of instructions executed. This is then related to the number of weights and architecture of the network where the complexity of the weights themselves also play a role. For instance, 64-bit floating point operations consume more power than 32-bit operations. Thus, if reduced floating point number schemes can be applied to reduce weight complexity, power consumption has the potential to be decreased.

[*]Equal contribution [1]Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA [2]Computer Science Department, University of Pittsburgh, Pittsburgh, Pennsylvania, USA. Correspondence to: None <NA>.

18-663: Hardware Architectures for Machine Learning, Second Project Report

1.3 Memory

One other potential benefit from reduced floating point approximate computing is a reduction in memory usage. Basic models contain thousands of parameters while advanced ones contain millions. For instance, ResNet-152 contains 60,344,232 parameters while AlexNet contains 62,378,344. This means memory for complex networks can be a bottleneck as the more time spent on data transfer between off-chip memory and on-chip cache, the slower training and inference is. In addition, running large models on edge devices can become in-feasible due to memory constraints. Thus, through the implementation of reduced floating point number schemes, edge devices can potentially be enabled to run ML applications that were previously impossible.

2 Related Work

There are various approximate computing schemes that can be applied to machine learning, particularly neural networks. Here we discuss several techniques, introduce related works, and examine accuracies for the MNIST (LeCun & Cortes, 2010) and CIFAR-10 (Krizhevsky et al.) data sets. In terms of Vivado HLS usage as a fast prototyping scheme for neural networks, only a single similar work was found.

One known approximation approach is weight discretization. A specific form of this type is Binarized Neural Networks (BNNs) (Courbariaux et al., 2016). In this technique, networks have binary weights and activations at run-time. At training-time the binary weights and activations are used for computing the parameter gradients. Another form of weight discretization is Ternary Weight Networks (Li & Liu, 2016). Here network weights are constrained to be +1, 0 and -1, ternary values, a balance between full precision and binary networks. These approximation methods can lead to faster, less resource constrained models, but can cause slight accuracy degradation.

Another well known quantization form is the use of fixed-point numbers. In the works of Lin et al. (2015) the authors experiment with various bit lengths on the CIFAR-10 data set and analyze the corresponding effect on accuracy. It is also well known that Google's TPU uses an 8-bit fixed-point scheme and clearly performs extraordinarily well on various model architectures and data sets.

In terms of Vivado HLS usage as a tool for FPGA synthesis and implementation of neural networks, in the works (Kreinar, 2017) the authors develop a library of pre-optimized C++ neural network functions. Here they use this library to create *IP Cores* which are then incorporated into lower level HDL using a wrapper to generate neural network compute engines.

For this project we train models and run experiments using two famous machine learning data sets, MNIST and CIFAR-10. In regard to MNIST, top neural network performance on the test set typically takes values from 97% to upwards of 99% accuracy (Ciresan et al., 2010) for MLP's. Meanwhile, CNN's typically score higher than 99%. In regard to CIFAR-10, top models currently achieve accuracies near 96% (Graham, 2014). The top performing CNN model has a reported accuracy of 95.59%(Springenberg). Unfortunately, with this being a harder data set to classify, MLP's perform poorly with top scores from 53% to 58%.

3 Technical Description

Here we describe the reduced floating point analysis conducted for each hardware platform (CPU, GPU, FPGA) and the tests implemented. Specific platforms used are as follow. Note that MNIST data set tests are conducted on CPU [1] whereas CIFAR-10 use CPU [2]. Also note that the same GPU and FPGA platforms are used for both data sets.

Platform	Specs
CPU [1]	Intel(R) Core i7 7660U 2.5GHz
CPU [2]	Intel(R) Core i7-8750H 2.20GHz
GPU	NVIDIA(R) GeForce GTX 1060
FPGA	Xilinx ZC706 Evaluation Kit

3.1 MNIST Neural Network Model Development

In regard to the MNIST data set, we develop five different neural network architectures for classification in PyTorch (Paszke et al., 2017). These networks consist of both multi-layer perceptrons (MLPs) as well as convolutional neural networks (CNNs). Likewise, these models have various numbers of hidden layers, filter counts, units in layers, activation functions, etc. to better analyze the effect on accuracy and the speed, memory, and power effects in hardware. PyTorch is used as it has automatic differentiation and builds a dynamic graph, thus making it unnecessary to compute backward propagation for each model by hand.

The architectures developed are as follow:

BasicNN: 784 inputs $\rightarrow$ 10 outputs (softmax activation)

BasicMLP: 784 inputs $\rightarrow$ 100 relu units $\rightarrow$ 10 outputs

AdvancedMPL: 784 inputs $\rightarrow$ 300 relu units $\rightarrow$ 100 relu units $\rightarrow$ 10 outputs

BasicCNN: 1x32x32 inputs $\rightarrow$ conv. layer (kernel = 5x5), filters = 6, relu act. $\rightarrow$ max pool (kernel = 2x2) $\rightarrow$ conv. layer (kernel = 5x5), filters = 16, relu act. $\rightarrow$ max pool (kernel = 2x2) $\rightarrow$ fully-connected layer (100 relu units) $\rightarrow$ 10 outputs

LeNet-5: 1x32x32 inputs $\rightarrow$ conv. layer (kernel = 5x5), filters = 6, relu act. $\rightarrow$ max pool (kernel = 2x2) $\rightarrow$ conv. layer (kernel = 5x5), filters = 16, relu act. $\rightarrow$ max pool (kernel = 2x2) → fully-connected layer (120 relu units) → fully-connected layer (84 relu units) → 10 outputs

3.2 CIFAR-10 Neural Network Model Development

Similar to as was done for MNIST, for CIFAR-10 we construct four neural networks consisting of both MLPs and CNNs with different structures in PyTorch.

The architectures are as follow:

BasicMLP: 3072 inputs → 100 relu units → 10 outputs

AdvancedMLP: 3072 inputs → 300 relu units → 100 relu units → 10 outputs

BasicCNN: 3x32x32 inputs → conv. layer (kernel = 5x5), filters = 16, relu act. → max pool (kernel = 2x2) → conv. layer (kernel = 5x5), filters = 16, relu act. → max pool (kernel = 2x2) → fully-connected layer (100 relu units) → 10 outputs

VGG-11 with Batch Normalization: Takes basic *VGG-11* architecture except all fully-connected layers are removed except for the final one to outputs

3.3 CPU Reduced Floating Point Analysis

The first series of tests conducted are in regard to the CPU platform where two varieties are run. In the first version, both training and inference for each model architecture is implemented using different floating point representations. This is fairly straight-forward as *torch.tensor* types can be specified to be any of three possible representations: 64, 32, and 16-bit floats. Unfortunately, 16-bit floats are unsupported with CPUs and for this test we only get results for 32 and 64-bits. In the second variety, trained 64-bit models are cast to 32-bits and the inference stage only is documented.

While running these tests, four statistics are recorded: speed, memory usage, power consumption, and final model accuracy. To measure speed, basic *time* statements are inserted into the Python code at certain places in the algorithm. Speed is analyzed both in regard to training time per epoch and inference time per sample. Memory usage is measured by recording the final model size in terms of bytes. CPU power consumption is documented using Intel's *Powertop* software. Similar to speed, power consumption is recorded both during training and inference. Finally, model accuracy is measured on the test set.

3.4 GPU Reduced Floating Point Analysis

After running the above experiments on CPU hardware, this procedure is repeated for the GPU platform. By using PyTorch to build models, transfer of these architectures to the GPU is straight-forward. CUDA, NVIDIA's GPU parallel processing language, is implemented into the framework of PyTorch thus allowing models and data to easily be transferred between a host CPU and a GPU. By sending architectures and data sets over to the GPU, experiments can then be run on it. This allows for categorization of the relative differences in terms of speed, memory usage, power consumption, and accuracy on separate hardware platforms.

An additional floating point scheme allowed on GPU hardware is the 16-bit floating point *half tensor*. NVIDIA GPUs fully support this number representation so the architectures and training performed for our CPU implementation is also implemented in *half precision* for GPU tests. Again, the tests ran consist of training models in *half precision* and then running inference on these learners, as well as casting trained *double precision* (64-bit) models down to 32 and 16-bits and running inference on them.

Similar to as in CPU tests, speed in regard to training and inference is again measured through insertion of Python *time* statements and memory usage is recorded as the final model size in terms of bytes. To track GPU power consumption, we use an NVDIA command-line tool called *nvidia-smi*. This tool allows GPU usage to be monitored and recorded with one of the metrics documented being power consumption. Finally, model accuracy is again measured for each data set's test set.

3.5 Vivado HLS FPGA Reduced Floating Point Analysis

After categorizing various model architectures on MNIST and CIFAR-10 with both CPU and GPU hardware, the next step was implementing these models on field programmable gate arrays (FPGA). FPGAs, being the most flexible hardware available, can be used to optimize every aspect of an algorithm. Unfortunately, one drawback to their usage is programming, in this case writing complex HDL code for neural networks. However, this gap was bridged through the use of Vivado High-Level Synthesis (HLS). With HLS, model architectures were implemented at a slightly higher level in C / C++ code and then synthesized directly as *IP Cores*. This allowed networks to be developed in a way that they could more easily be modified in regard to hyper-parameters and architecture than in strictly HDL format.

In this experimentation we dealt with multi-layer perceptron (MLP) architectures only. The reasoning behind this was that we focused on the optimization of the HLS code and minimized inference latency as much as possible while increasing FPGA resource utilization. If we had focused development on two separate fronts in regard to developing a framework for both MLPs and CNNs we most likely wouldn't have been as successful in regard to achieving low latency times.

The first step of developing our MLP implementations for

FPGAs on HLS was writing a C-level framework. This basic framework was developed to allow massive flexibility in regard to architectural modifications. This was done by building various neural network blocks such as different activation functions (relu, sigmoid, tanh, etc.) and a basic linear layer structure. After creating these blocks, each was augmented with potential optimizations that result in decreased latency while increasing system resource utilization. These optimizations, referred to as HLS *pragmas*, will be discussed shortly.

Building this framework with these basic structures was necessary as we had five different MLP architectures to create from earlier. Four of these architectures had three separate instances with each using a different floating point scheme. Therefore, the end goal in regard to this framework was to develop code allowing models to be specified nearly as easily as in PyTorch with the floating point representation being straight-forward to modify.

After developing the basic C / C++ framework, each of the MLPs developed earlier was built. As these models were trained during CPU / GPU analysis, network parameters were uploaded into the FPGA directly as inputs to allow for categorization of the inference stage. After building these models in HLS, designs were synthesized to hardware and exported as *IP cores*.

To test for Vivado HLS model accuracy, the built-in C-simulator was used. C code was compiled and a test-bench was developed feeding test set samples into the network. Due to large test set size, testing all data points sequentially and calculating total accuracy proved difficult due to segmentation faults. The work-around we implement is to use a subset of the test set. Here we take the first one hundred samples and calculate the accuracy on them. This is then compared to the accuracy over the same samples in our PyTorch implementation to ensure accuracy values match. After this, Vivado HLS's C / RTL co-simulator is used to again ensure test-benches pass.

After exporting HLS designs as *IP Cores*, the last step was to incorporate these cores into low-level HDL code. This process was done using Vivado's core environment and their *IP Integrator* tool-kit. By using this tool-kit we were able to quickly connect neural network *IPs* to a system clock and input / output ports to buses, completing the overall design. This design was then synthesized and implemented for our specific FPGA hardware. After running these processes we analyzed output reports to categorize latency, power, and resource utilization statistics.

One added aspect to HLS model implementation is the addition of hardware level *pragmas*. These are statements that can be added directly to the C / C++ code that can specify how to utilize the available hardware making up the FPGA. For example, *for loops* can be parallelized, arrays of memory can be partitioned to allow for greater numbers of concurrent reads / writes, data movement can be pipe-lined, etc.

The core operation we focused on optimizing was the matrix multiply-accumulate, or MAC, operation. This is the basic computation in linear layers and the bottleneck in regard to fast latency times. Optimization was done by pipe-lining computation, unrolling the *for loop* that cycles through all output neurons (as they can be processed in parallel), and partitioning input arrays and weight matrices to allow for greater number of concurrent reads / writes thus allowing increased multiplications in parallel. Finally, *adder trees* were hand implemented within *for loops* as sequential addition proved to be another latency bottleneck.

3.6 Vivado HDL FPGA Reduced Floating Point Analysis

To explore the effectiveness of fully customized implementation of neural networks, we then tried implementing one of the Basic MLP networks on MNIST using Verilog. To do this, we exploit Xilinx IP Cores for floating point operations. Here we developed a node that gets weights, data points, bias values, and a clock signal as inputs and outputs the y value. In each node we dedicated enough floating point multipliers to calculate $w_i \times x_i + b_i$ for each input. All multiplications can occur in parallel, however, summing the results is the bottleneck and needs to be done sequentially.

To improve the latency of the summation part, we devised an adder tree such that we dedicated 10 levels of floating point adders. Moreover, to improve the throughput, we placed a set of registers after each multiplication and each level of additions. In essence, we implemented each node in a pipelined fashion. To implement the neural network we instantiated this node in a top level module for the number of nodes in each hidden layer in the Basic MLP. Finally, to make a decision we used a floating point comparator IP core to find a maximum probability among the 10 output classes.

4 Results

For the MNIST data set we train all models for five epochs except *LeNet-5* which is only trained for four. The optimizer used is Adam with a learning rate of 0.001. Note that MLP models were trained with a batch size of 32 whereas CNN models used a batch size of 128.

In regard to CIFAR-10, all models were trained over 100 epochs using a batch size of 32. Additionally, the optimizer used is stochastic gradient descent (SGD) with 0.001 and 0.9 as the learning rate and momentum value respectively. Note that CPU models were not trained for *VGG11BN* due to long training time.

4.1 Test Set Accuracy

The following tables display test set accuracy values for different models with different bit-widths. Note that accuracy values from casting 64-bit models to 32 / 16-bits are within 0.02% of original values.

4.1.1 MNIST

CPU Model	64-Bit	32-Bit
BasicNN	NA	92.79%
BasicMLP	97.6%	97.28%
AdvancedMLP	97.65%	97.58%
BasicCNN	98.53%	98.52%
LeNet-5	98.3%	98.51%

GPU Model	64-Bit	32-Bit	16-Bit
BasicMLP	97.5%	97.71%	97.27%
AdvancedMLP	97.71%	97.81%	98.05%
BasicCNN	98.6%	98.57%	98.18%
LeNet-5	98.67%	98.64%	98.56%

In regard to FPGA accuracy values, the first one hundred test set samples are used and accuracy is verified to match that of the PyTorch implementation. The CPU versions of 32-bit **BasicNN**, 64 / 32-bit **BasicMLP**, and 64 / 32-bit **AdvancedMLP** are implemented on the FPGA. Likewise, the 16-bit GPU versions of **BasicMLP** and **AdvancedMLP** are also implemented.

4.1.2 CIFAR-10

CPU Model	64-Bit	32-Bit
BasicMLP	52.69%	52.94%
AdvancedMLP	54.50%	54.81%
BasicCNN*	65.28%	65.00%

GPU Model	64-Bit	32-Bit	16-Bit
BasicMLP	52.77%	52.41%	52.38%
AdvancedMLP	54.85%	54.79%	54.66%
BasicCNN	65.78%	67.26%	65.80%
VGG11BN	79.19%	79.16%	79.18%

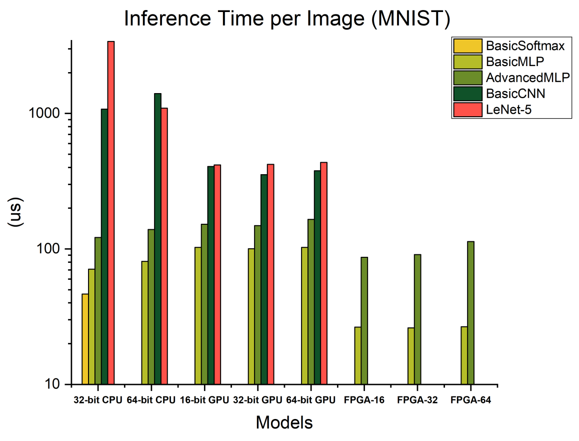
*Note: * indicates model only trained for 19 epochs*

In regard to FPGA accuracy values, again the first one hundred test samples are used and accuracy is verified to match that of PyTorch. The CPU versions of 64 / 32-bit **BasicMLP** and 64 / 32-bit **AdvancedMLP** are implemented on the FPGA. Additionally the 16-bit GPU versions of **BasicMLP** and **AdvancedMLP** are also implemented.

4.2 Inference Latency

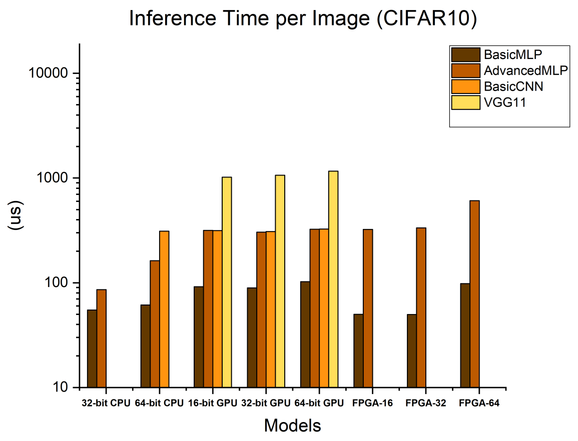
Inference latency per image for different architectures and floating point representations are as follow. Note that the batch size is set to one for all platforms and architectures (why there is no significant improvement between CPU and GPU times in some cases). For consistency these times only consider the forward pass and output prediction, ignoring data movement and transfer within the hardware platform.

4.2.1 MNIST



4.2.2 CIFAR-10

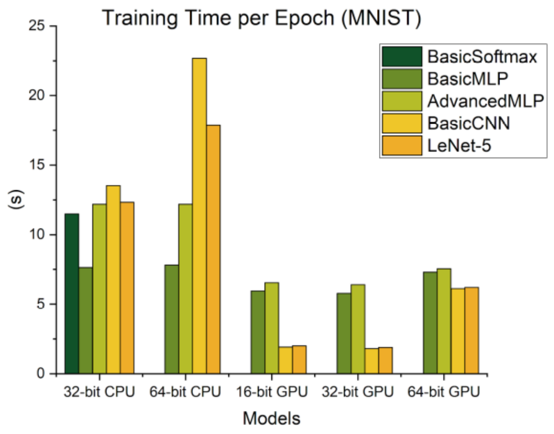
Note that times are not reported for the 32-bit CPU version of **BasicCNN**. This model experienced a memory leak when training and thus inference time results were unreliable. Also note inference times are not shown for the CPU implementation of **VGG11BN** as they were significantly higher than all other results due to the complexity of the model. The 64-bit version took 16,920us per image while the 32-bit version took 11,793us.



4.3 Training Speed

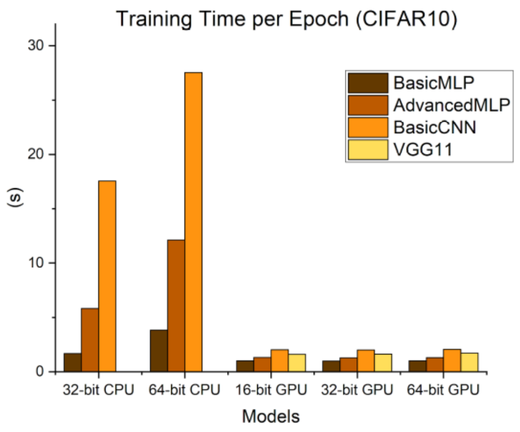
Training time per epoch for different architectures and floating point schemes are as follow. Time is only recorded for the forward / backward pass and during parameter updates for CIFAR-10. On the other hand MNIST results include reading in data and data transfer along with all neural network computation. Note that FPGA implementations were used for inference only and thus not shown.

4.3.1 *MNIST*



4.3.2 *CIFAR-10*

Note that CPU platform training time per epoch for *VGG11BN* was 517.94 seconds for 32-bits and 1,614.04 seconds for 64-bits. This model was not trained for all 100 epochs due to long run times.



4.4 Memory Requirements

Memory requirements for different network architectures and floating point number schemes are as follow. Here memory consumption is measured as the number of bytes a saved model takes up. As there is no difference in this size based on platform, only a single value is reported for each configuration.

4.4.1 *MNIST*

Model	16-Bit	32-Bit	64-bit
BasicNN	NA	31.9kB	NA
BasicMLP	159.8kB	318.9kB	636.9kB
AdvancedMLP	534.3kB	1.1MB	2.1MB
BasicCNN	88.8kB	176.2kB	350.9kB
LeNet-5	125.2kB	248.6kB	495.4kB

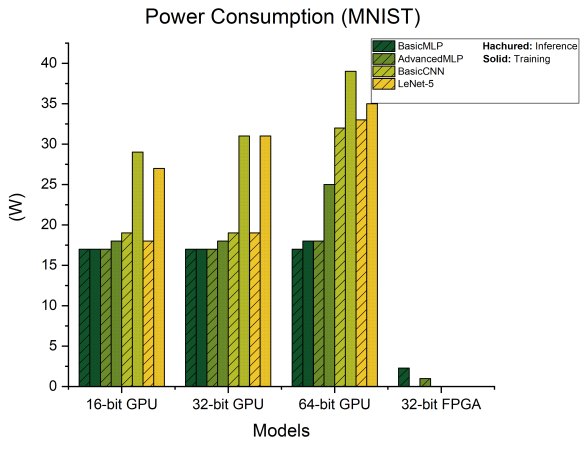
4.4.2 *CIFAR-10*

Model	16-Bit	32-Bit	64-bit
BasicMLP	617.5kB	1.2MB	2.5MB
AdvancedMLP	1.9MB	3.8MB	7.6MB
BasicCNN	99.0kB	196.4kB	391.4kB
VGG11	18.5MB	37.0MB	73.9MB

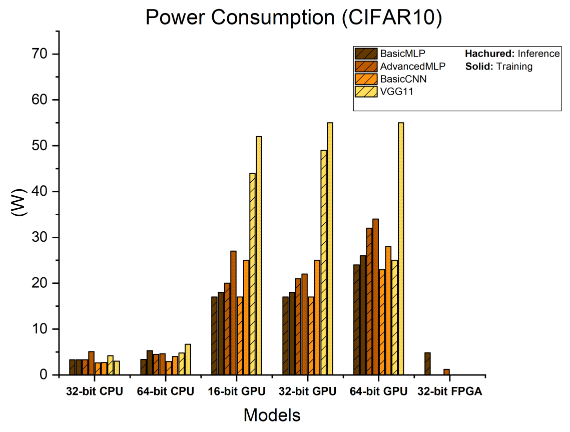
4.5 Power Consumption: Training and Inference

Training and inference power consumption for different model architectures and floating point schemes on various hardware platforms are as follow. Note that CPU power statistics are not recorded for the MNIST data set.

4.5.1 *MNIST*



4.5.2 *CIFAR-10*

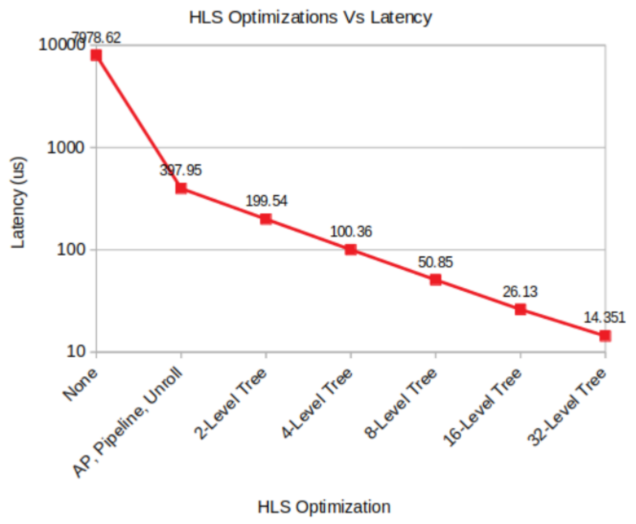


4.6 Vivado HLS FPGA Optimization

Results regarding latency and system resource utilization while implementing various HLS *pragmas* to optimize hardware are as follow. The *pragmas* considered are pipelining, loop unrolling, and array partitioning. We also analyze the effect of *adder trees* as another optimization technique. This was implemented for the **BasicMLP** model using 32-bit floats on the MNIST data set.

Note: "All" for optimization refers to pipelining, unrolling, and array partitioning; Following is adder tree depth

Optimization	DSP Usage	FF Usage	LUT Usage
None	1%	≈ 0%	≈ 0%
All	4%	6%	17%
All; 2-Level	8%	8%	20%
All; 4-Level	12%	9%	23%
All; 8-Level	20%	12%	29%
All; 16-Level	36%	19%	38%
All; 32-Level	71%	31%	60%



4.7 Vivado HLS 8-Bit Fixed Point

The following resource utilization stats are from using an 8-bit fixed point number scheme. This model was not trained or tested, hardware was simply converted to this format in HLS and FPGA utilization stats were collected.

Optimization	Latency	DSP	FF	LUT
All; 16-Level	25.61us	0%	3%	41%

4.8 FPGA HDL Implementation

Regarding the FPGA Verilog implementation, we faced several obstacles preventing us from completing the implementation, thus we do not have results for this. One issue we dealt with is that 2-dimensional arrays are not allowed to be defined as an input of a module. Thus, to pass the weights to a node, we had to flatten all the signals then reconstruct the array inside each node. The reason we tried the HDL implementation is that 1) we wanted to investigate the affect of a custom low-level implementation on utilization and latency on FPGA. 2) With the HDL implementation we had the chance to lower the bit width to 8. Xilinx floating IP cores can be configured to 8-bit and we also defined the module parametric so that upon completion it will be easy to sweep over bit width from 64 to 8.

5 Analysis

Looking at our results we have a number of takeaways. First, reduced floating point computation in 16-bit precision performs as well in regard to accuracy as 32 and 64 bits. This is therefore the optimal floating point scheme to implement due to reduced model storage capacity, training time, and power consumption. Additionally, model size is proportional to the number scheme used. Going from 16-bits to 32-bits doubles the size as does going from 32 to 64.

Another takeaway, and as expected, training and inference are significantly faster on GPU platforms compared to CPUs. As the model size grows, this relative difference increases. However, there is no noticeable difference in training or inference time between 16, 32, and 64 bits on a GPU. This can be attributed to the way computation is mapped to the hardware. On the other hand, when running training and inference on a CPU platform, it is best use the 32-bit scheme as it results in faster computation.

Regarding power consumption between CPU / GPU platforms, GPUs consume significantly more power, often on the scale of 5x. Looking closer at these numbers, training appears to consume more power than inference. This is due to the increased level of complexity from back-propogation computation. Comparing power consumption between 32 and 64 bits on CPUs, 32-bits is more power efficient. On the other hand, when comparing the values between 16, 32, and 64 bits on GPUs there is no noticeable difference between 16 and 32 bits. However, as the number scheme is increased to 64-bits power consumption jumps.

In terms of FPGA analysis we can immediately see the massive potential for hardware level optimization in machine learning applications. In nearly all cases we are able to achieve a shorter latency and lower total power consumption than both CPU and GPU platforms. This was due to the fact that we were able to increase parallel computation based on the simple and repeated nature of MLP operations.

Again, in the above FPGA experimentation we focused on optimizing for latency. Thus we sought to utilize as much hardware as possible for fast computation. Another direction we could have chosen to take, and a trade-off afforded due to FPGA flexibility, is we could have optimized for power consumption. This would have been done by utilizing fewer system resources while allowing for slower computation speed. These trade-offs between latency, power consumption, and resource utilization provided to developers on FPGA platforms allow designs to be created based on the constraints at hand.

One final takeaway is the potential of Vivado HLS as a fast development tool for deep neural network prototyping and development. As we quickly learned, developing these systems at the lower HDL level in Verilog comes with numerous complications. There are enormous amounts of total computation, lots of data movement overhead, and less generality as modifying network architecture can be difficult. On the other hand, by using Vivado HLS we were afforded a much higher level of abstraction. In terms of using different floating point number schemes on the FPGA, its best to go with 16-bits versus 32 or 64. By using 16-bits fewer system resources are used thus allowing more optimization to be done. This could be extended even farther when implementing fixed point numbers or even binary networks.

6 Schedule and Milestones

6.1 Milestone 1: PyTorch CPU Implementation

Milestone Date: Completed

Delivery: Source code for training / inference, architectures, and 64 and 32-bit float training / inference stats

Team Member: Mehrnoosh: MNIST - Dillon: CIFAR-10

6.2 Milestone 2: PyTorch GPU Implementation

Milestone Date: Completed

Delivery: Source code for GPU training / inference and 64, 32, and 16-bit float training / inference stats

Team Member: Mehrnoosh: MNIST - Dillon: CIFAR-10

6.3 Milestone 3: Vivado HLS Implementation

Milestone Date: Completed

Delivery: C / C++ source code for generic MLP architecture

Team Member: Dillon

6.4 Milestone 4: IP Core Implementation of Models

Milestone Date: Completed

Delivery: Exported IP cores for MLP model architectures

Team Member: Dillon

6.5 Milestone 5: IP Core Model FPGA Test

Milestone Date: Completed

Delivery: 64, 32, and 16-bit float inference stats MNIST / CIFAR-10

Team Member: Dillon

6.6 Milestone 6: Custom Verilog HDL Implementation

Milestone Date: Some progress

Delivery: Source code for implementation

Team Member: Mehrnoosh

6.7 Milestone 7: Testing of Custom Verilog Implementation

Milestone Date: Did not complete

Delivery: Inference stats on MNIST / CIFAR-10 (Based on model implemented)

Team Member: Mehrnoosh

References

Ciresan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. Deep big simple neural nets excel on handwritten digit recognition. *CoRR*, abs/1003.0358, 2010. URL http://arxiv.org/abs/1003.0358.

Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1, 2016.

Graham, B. Fractional max-pooling. *CoRR*, abs/1412.6071, 2014. URL http://arxiv.org/abs/1412.6071.

Kreinar, E. Rfnoc neural network library using vivado hls. *Proceedings of the GNU Radio Conference*, 2(1): 7, 2017. URL https://pubs.gnuradio.org/index.php/grcon/article/view/27.

Krizhevsky, A., Nair, V., and Hinton, G. Cifar-10 (canadian institute for advanced research). URL http://www.cs.toronto.edu/~kriz/cifar.html.

LeCun, Y. and Cortes, C. MNIST handwritten digit database. 2010. URL http://yann.lecun.com/exdb/mnist/.

Li, F. and Liu, B. Ternary weight networks. *CoRR*, abs/1605.04711, 2016.

Lin, D. D., Talathi, S. S., and Annapureddy, V. S. Fixed point quantization of deep convolutional networks. *CoRR*, abs/1511.06393, 2015. URL http://arxiv.org/abs/1511.06393.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. 2017.

Springenberg. 1412.6806.pdf. https://arxiv.org/pdf/1412.6806.pdf. (Accessed on 04/04/2019).