High-Performance Computing:

Optimizing Performance for Optical Character Recognition via Neural Networks

Takao Yamada (tyamada@bu.edu) Graduate Electrical & Computer Engineering

Mikhail Andreev (mikh@bu.edu) Graduate Electrical & Computer Engineering

May 8, 2015

Table of Contents

Abstract

Neural networks have been shown to have the ability to perform character recognition given a set of training images. Using these concepts, we seek to optimize the code execution using hardware capabilities of a CPU and GPU. Using techniques including loop unrolling, threading, and GPU partitioning, we analyze the effectiveness of these methods in utilizing a larger percentage of the hardware to perform expensive calculations.

Introduction

1. Character Recognition

Optical Character Recognition involves reading in images of characters and classifying them as a given character. This is done by looking at the features present in the character images and comparing them to the features present in training samples. After enough comparisons are made, the new image can be designated as a specific character. The method with which we perform the classification is through the use of a neural network.

2. Neural Networks

Neural networks act as interconnected graphs modeling neural activity found in our brain. The neurons are arranged in layers, which are connected to each other through edges. The edges have a specific weight associated with them. This weight forms the associative connection between neurons of different layers. By training the neural network, the weight of this connection will increase or decrease based on the association of that particular feature with the desired result. As a simplistic example, we can consider that each neuron calculates a particular feature of the input image, and then compares it to the same types of features present in the training images. If this feature is a good indicator of how the input image should be classified, the weight of that edge will be strengthened. Otherwise, it will be weakened and that feature will have a smaller impact on the image classification.

The anatomy of neural network consists of several layers, each of which has neurons in it. The first layer is known as the input layer, and the last layer is the output layer. Every other layer is a hidden layer which performs further computations. Each neuron in a layer has a weighted edge coming to it from every neuron in the previous layer, and has an edge going to every neuron of the following layer.

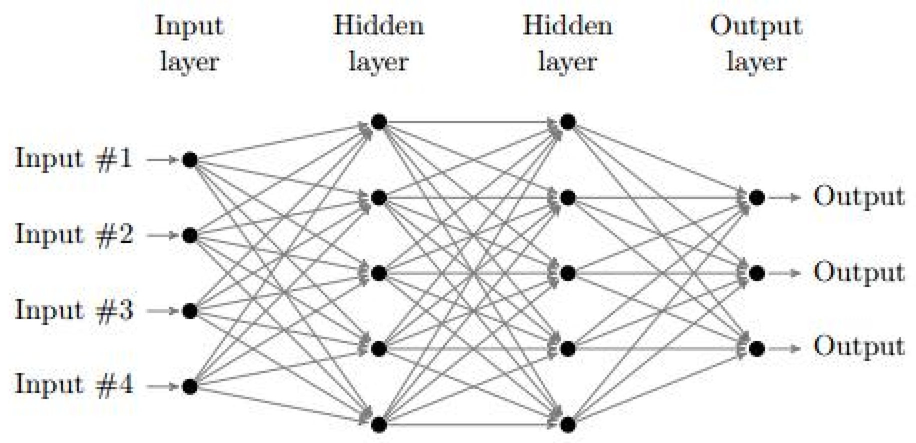


Figure 1: Topology of a simple neural network

Images are passed though the network using a feed-forward mechanism which performs a calculation based on the weights of a node and then passes the results on to the next layer. These calculations can be summarized with the equation:

$$b_j = \sum_{k=1}^{m} w_{k,j} a_k \; a_j = \sigma(b_j)$$

m is the number of neurons in each hidden layer, $w_{k,j}$ is the weight of an edge (k,j) and a_k is the input to the system. σ stands for the sigmoid function, used to approximate the activation threshold:

$$\sigma(x) = \frac{1}{1 + e^{-x/a}}$$

To train the network with input samples we use a method called back-propagation. This involves first calculating the feed-forward results of the system. After this, since we know the correct result, we compare that with the calculated value, and use the error generated through that calculation to adjust the weights of the edges starting with the output layer, and propagating backwards through the system.

3. Method

Our method for optimizing the performance of the neural network is to use techniques that allow the available hardware to be fully utilized to perform calculations.

A. Base Case

Our focus for optimizations involved the feed-forward and back-propagation functions. For the base case these involve calculating summations and performing the sigmoid function. Each of these operations is done for each layer, and the values used by a layer depend on the values that come from the previous layer. However, within a single layer these values are independent for each neuron, making that the ideal place to insert optimizations.

a. Single Core Optimizations

We began optimizations by assuming the use of a single core. To optimize under these conditions we had to use the super-scalar functionality of the processor. This involved the use of loop unrolling and multiple accumulators. Loop unrolling was effective in this case since most of the computations involved were performed through the inner-most loop of the functions. By unrolling the loops to a factor of x12 we were able to maximize performance gains. The majority of these gains arose from decreasing loop overhead, eliminating branch prediction, and allowing for instruction-level parallelism.

To further benefit instruction-level parallelism, we introduced multiple accumulators into the loops. This allowed for the processor to compute multiple unrolled operations at a faster rate thanks to pipelining the instructions. Since the data dependence of the accumulator was removed from the operations, this pipelining was allowed reach its full capacity.

b. Multiple Core Optimizations

By utilizing multiple cores on the CPU we were able to achieve a new level of performance boost by parallelization via threading. We achieved this using the OpenMP and pthread constructs. These two techniques allow code to be run on multiple threads, allowing several functions to be executed in parallel. OpenMP parallelism was used to parallelize the inner loop among several threads. Meanwhile, pthreading performed a work function that replaced the inner loop.

The positive effects of threading can be significant, potentially decreasing execution time by the number of threads used. However, this case is only possible when there is a large input size. If the input size is too small, the overhead from running these methods greatly outweighs any benefits they may give.

c. GPU Optimizations

In a simple approach to GPU implementation, each GPU thread can represent a neuron in a layer. Since each neuron in a layer is independent of each other, this makes parallel execution seem like an ideal solution. The calculation is a matrix vector multiplication (MVM) and can be easily optimized. The MVM is calculated for each input and on every layer of the neural network. However, since the number of neurons differs within each layer, the MVM changes size with every iteration. This means allocation and cleanup of memory in both the CPU and GPU differs constantly. There is plenty of overhead due to this and in the results it can make a huge difference in performance especially for smaller sample sizes.

Note, that the GPU used in this project had to be a single precision float instead of the double precision that is detailed in the original code. If the GPU in the lab allowed double precision, then the results may vary.

In further optimizing the code, the GPU implementation can be changed to break the problem down using tiling, shared memory, and coalescing. By breaking the matrix into smaller problem size, the data can all fit into shared memory which is faster to grab data. With coalescing, the threads will grab data from shared memory together in columns instead of rows. This allows each thread to retrieve data from each row after a cache miss instead of having many threads overlapping cache retrievals.

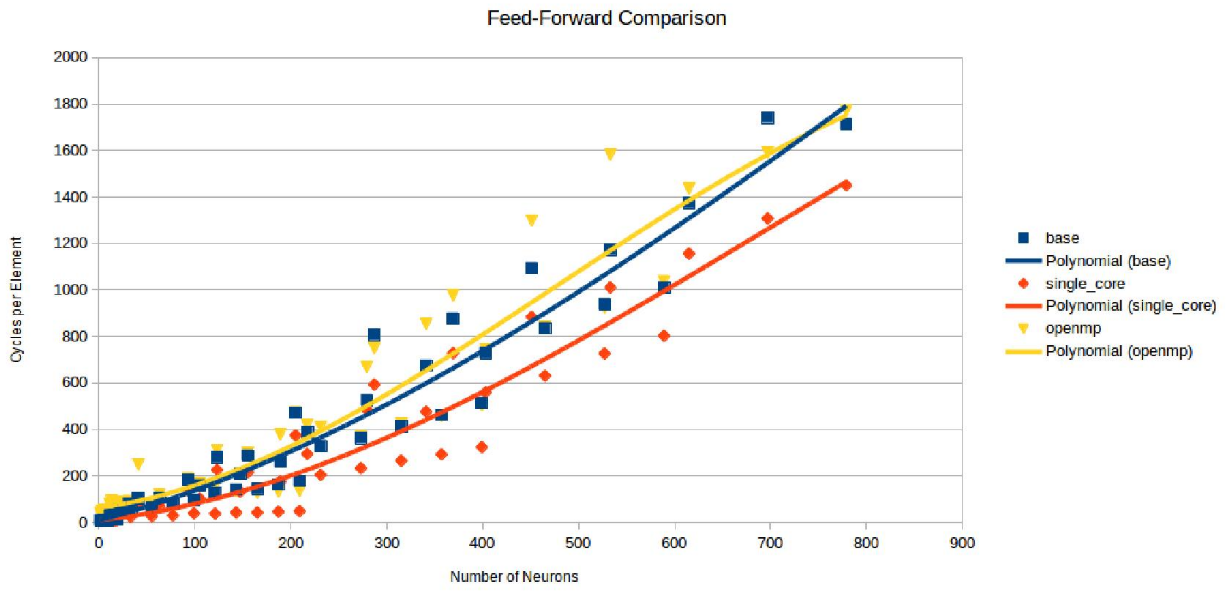
The worst case scenario for GPU implementation is when the overhead takes significantly longer than the actual calculation itself. And this is exactly what we faced in trying to optimize the code. The input size is fixed at 6 double precision by 6 double precision images and the output layer is only a few neurons long. This leaves each MVM calculation at the beginning and at the end with small sample sizes. If there were too few layers, the memory allocation and cleanup and data transfers takes up enough time that it takes longer to run a GPU optimized code than the single core with no optimizations. In the hidden layers the MVM calculations are just as big as the number of neurons at each layer. This makes each hidden layer MVM calculation on the GPU more efficient than the single core no optimization.

Results

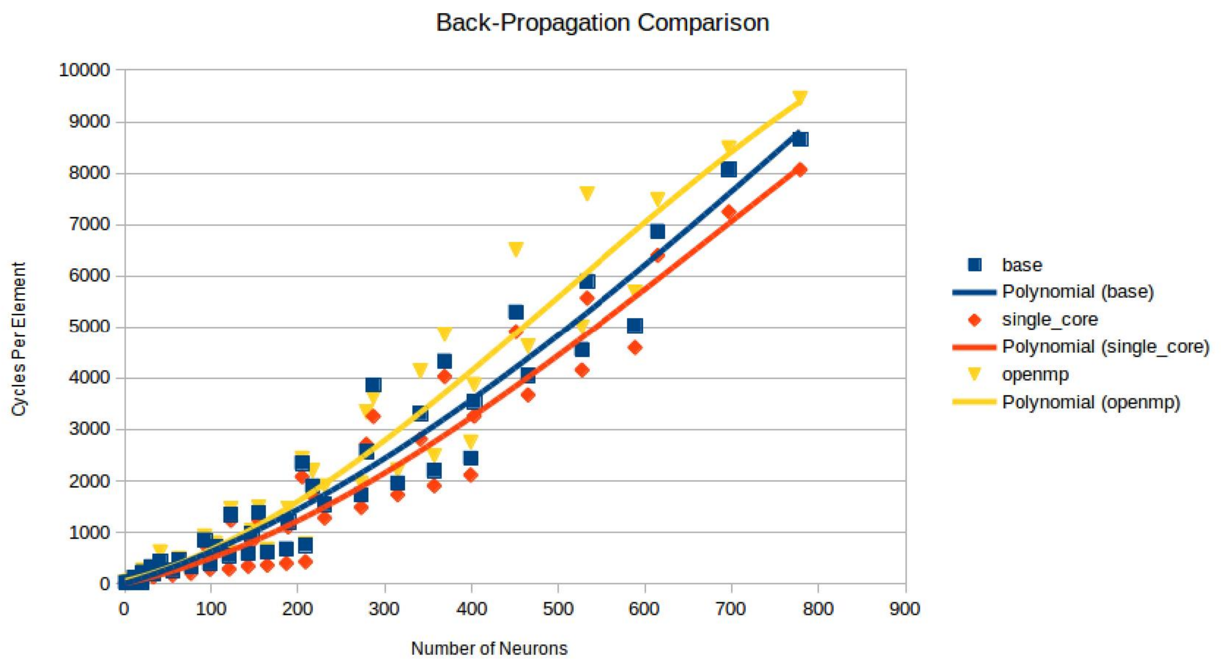
CPU Implementation

The results for the CPU optimizations were varied. The single core optimizations showed consistent improvement over the base case. However, the speedup due to it was not very high, and as sample size increased, the speedup stayed constant.

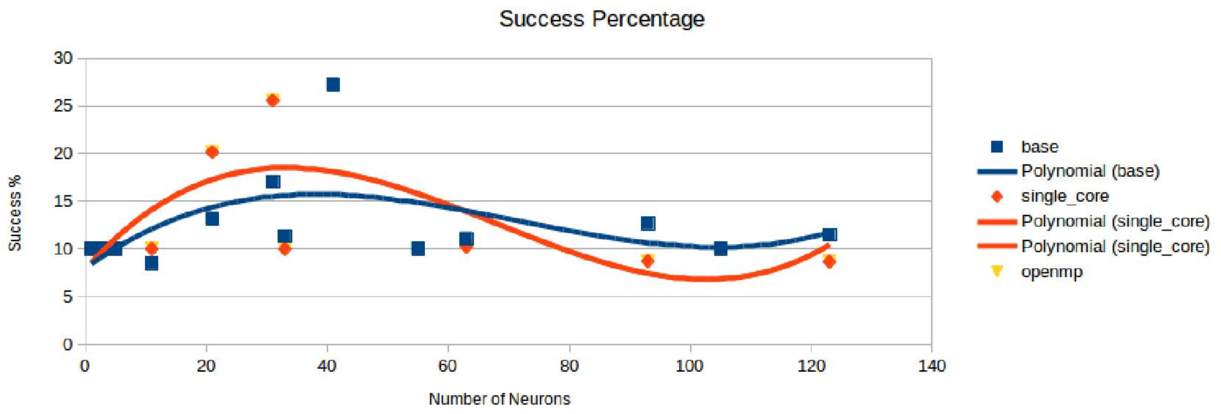
Unfortunately, when we utilized OpenMP and pthreads, performance decreased drastically. The overhead associated with using these methods proved to be too large to allow for speedup gains. It can be seen in the graph below that as input size increases, OpenMP manages to match the speed of the base case, indicating that if it were to keep using a larger input size, OpenMP could give appreciable gains to the computation speed.



For the back-propagation side, we can see the results are much closer together for different optimizations and the CPE values are much higher. This is due to the nature of the code which allowed for much less optimization without affecting results.



Based on the results, we can see the percentage of successful classifications based on number of



neurons. As the number of neurons increases the success percentage drops drastically.

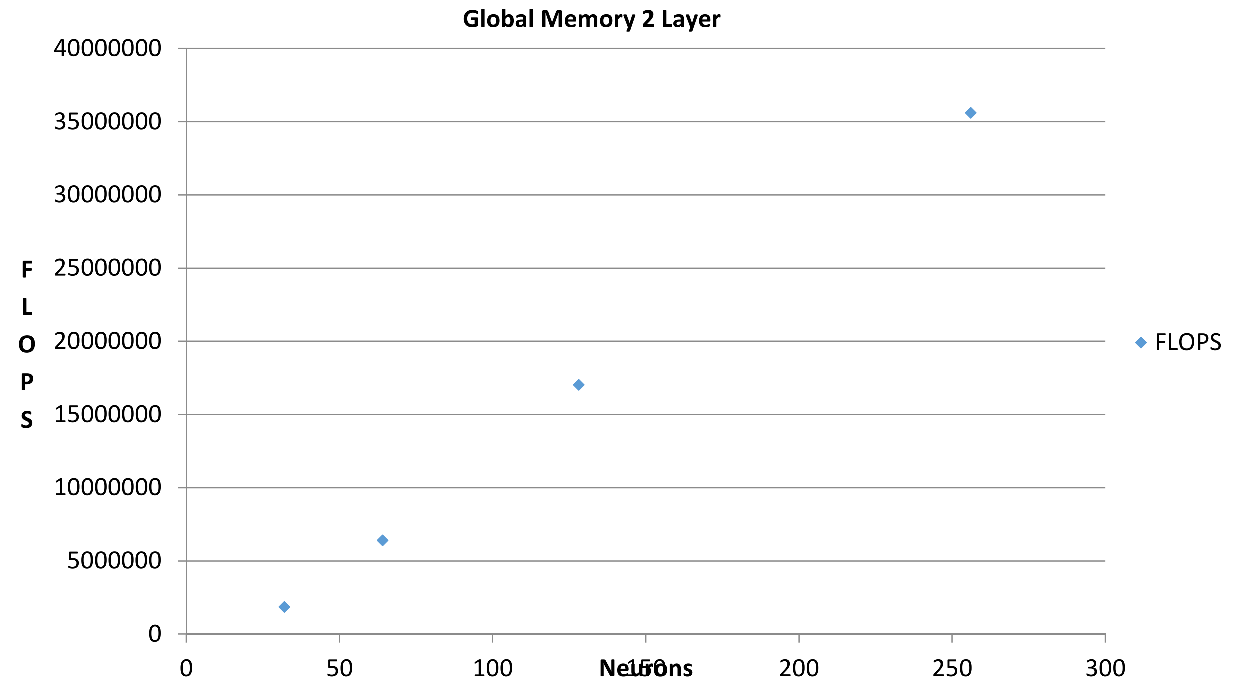
GPU Implementation

For a MVM calculation, there are 2 floating-point operations per iteration and Matrix Size * Matrix Size iterations of the loop body. Due to the matrix and vector size changing at each iteration, it is difficult to make an exact GFLOPS calculation. The best solution was to take the time to do the whole feed forward and whole back-propagation times and use the maximum Matrix size dependent on the number of neurons. For feed forward and back-propagation, MVM calculations are done on every layer for every image. Since there are 10 characters to recognize with 1000 images each, that is a total of 10,000 inputs. For global memory 4 layers, the extra iterations were by 4 while the rest were by 2.

At 512 neurons, the GPU seemed to stop working so only up to 256 was calculated in the following tables and graphs.

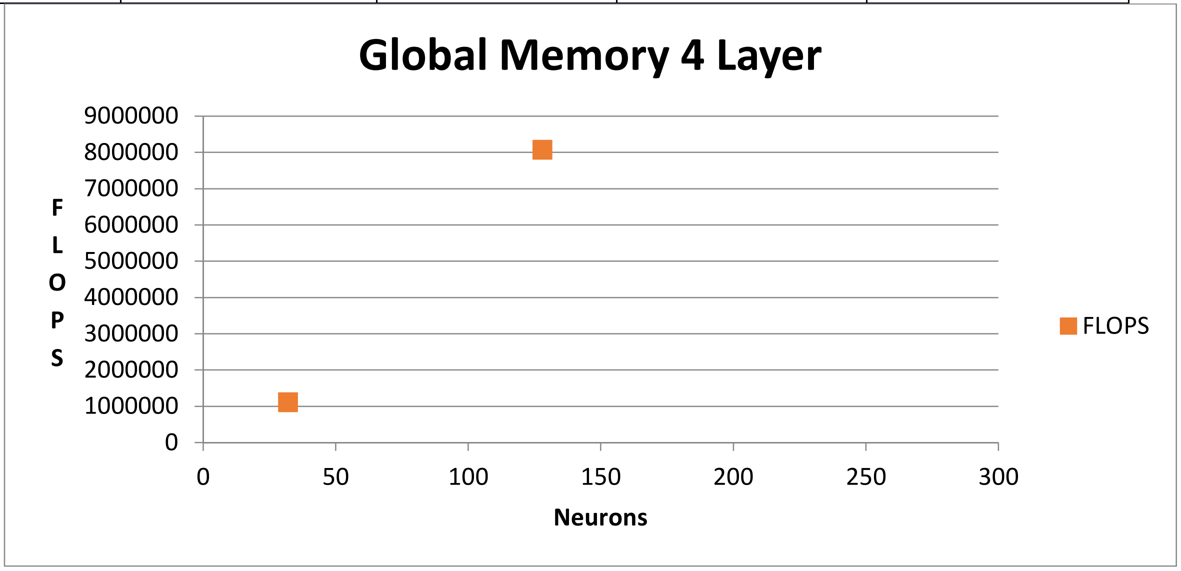
1. Global Memory on 1000 images per character on 2 layers

Neurons	Forward (ms)	Back (ms)	Accuracy	FLOPS
32	22109.22656	22711.46875	19.25%	1852620
64	25563.23047	18580.32422	18.85%	6409205
128	38487.08984	24335.51758	16.46%	17028047
256	73603.24219	30462.10742	9.62%	35615822



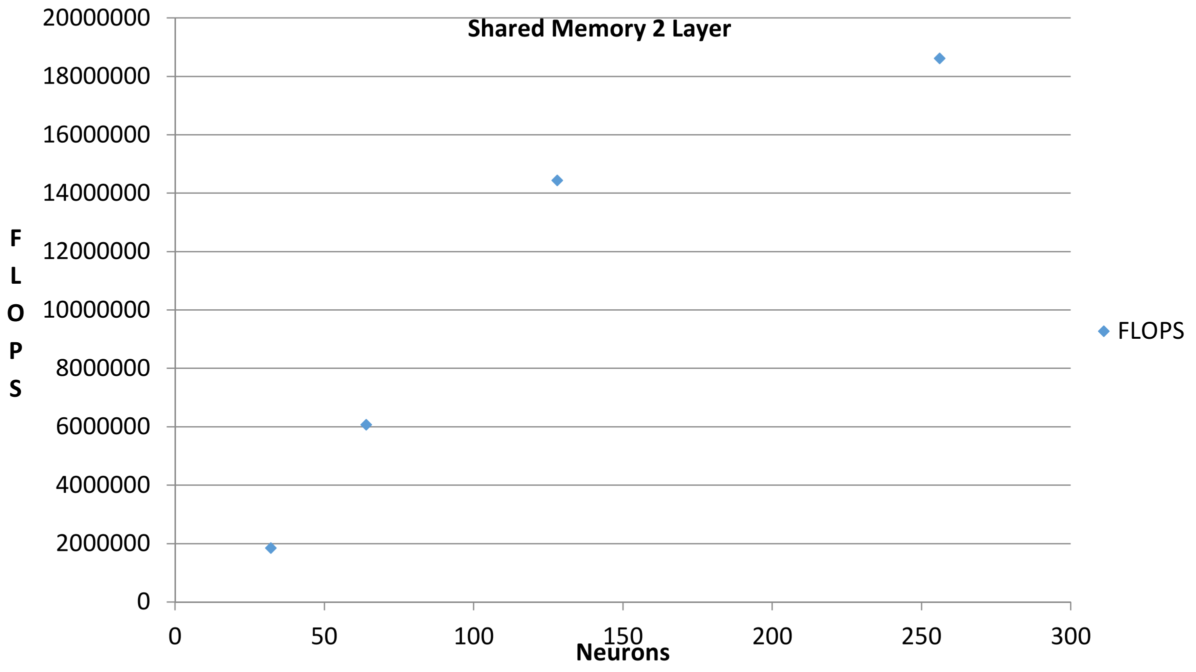
2. Global Memory on 1000 images per character on 4 layers
   Note, only 2 calculations were done because accuracy got too low.

Neurons	Forward	Back	Accuracy	FLOPS
32	36598.25391	29046.52148	10.20%	1119179
128	81108.6875	45087.51563	9.70%	8080022



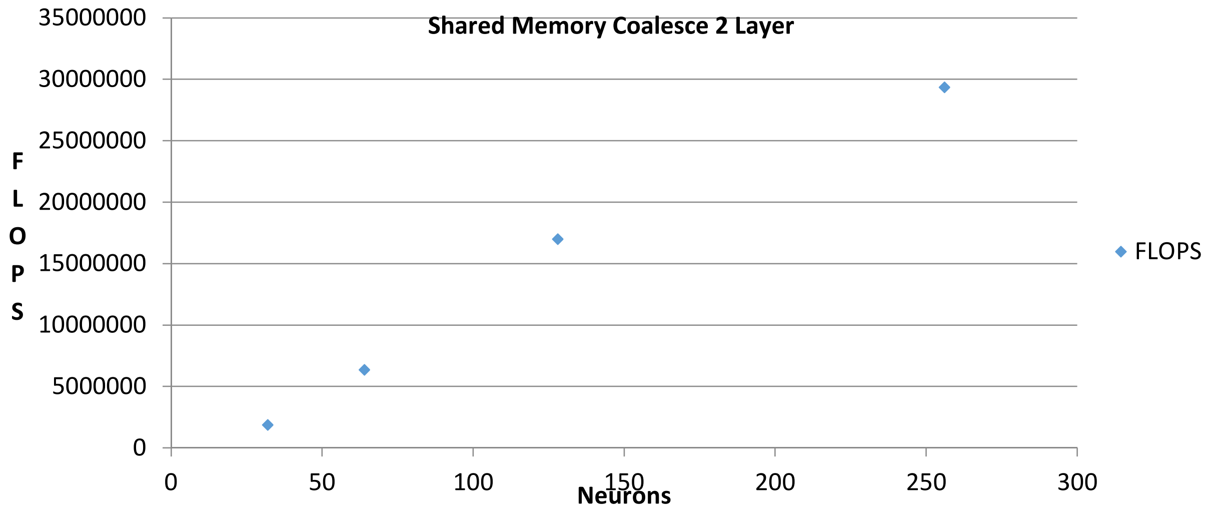
3. Shared Memory on 1000 images per character on 2 layers

Neurons	Forward (ms)	Back (ms)	Accuracy	FLOPS
32	22135.76758	17962.28516	17.65%	1850398
64	26975.05664	20255.61719	17.19%	6073759
128	45393.19141	30095.2207	19.60%	14437407
256	140820.5469	91786.05469	16.62%	18615465



4. Shared Memory with coalescing on 1000 images per character on 2 layers

Neurons	Forward (ms)	Back (ms)	Accuracy	FLOPS
32	21894.35547	17709.69336	10%	1870801
64	25791.17188	19252.06445	18.85%	6352561
128	38557.66797	24388.73633	15.44%	16996878
256	89314.89063	49809.20703	16.16%	29350536



Future Work

For GPU implementation, it would be much more efficient if the input sizes were much bigger. This would improve the MVM calculations at every layer. However, not much more parallelism can be made when each input must go through the feed forward and back propagation before the next input can be processed which rules out any pipelining. Even on the layer level, each layer must calculate the MVM before the next layer and pipelining would affect the previous and next iteration of the input.

A double precision GPU would greatly improve the accuracy of the network and prevent lost data from initial layers to propagate further into the network. With the weights in each node being within a small range, every bit of precision can make a big difference.

Conclusion

If the sole purpose of this project was to show the speed up of using all the optimization methods, then the results show that the optimizations in single core and multicore does improve as the number of neurons and layers increased. In GPU, the FLOPS column tells us that as the matrix size increases the GPU was performing more floating point operations per second.

However, the neural network is only useful if its accuracy and success rate was high. What we found is that the higher number of neurons and layers actually made the success rates to decrease. There is saturation in the hidden layers of the network due to the use of the sigmoidal function in back-propagation. This means the output of the neural network is almost constant for most of the input patterns. When going through the back-propagation to fix the weights, the information is lost somewhere in the multiple layers and vast number of neurons. This can essentially make back-propagation useless and exhibit the same behavior of a non-learning neural network. On the other hand, having a large database of inputs but with too few neurons and layers will have problems preserving the information in the network. There is a delicate balance between the number of inputs, number of neurons, and number of layers for the neural network to function.

As we found, the GPU or multicore implementation had performance decrease when the neuron count was low, however this meant the neural network's accuracy was much better. With bigger neural networks the GPU and multicore improved the performance overall, but the accuracy of the neural network was abysmal. In the end, the best optimization will highly depend on the parameters set when running the neural network.

References

Base code of "Neural Network OCR", https://github.com/bcuccioli/neural-ocr, May 7, 2015.

"A regularization term to avoid saturation of the sigmoids in multilayer neural networks". http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.41.1741&rep=rep1&type=pdf. Lluis Garrido, Sergio Gomez, Vicens Gaitan, and Miquel Serra-Ricart.