High-Performance Computing:

Optimizing Performance for Optical Character Recognition via Neural Networks

Takao Yamada ([tyamada@bu.edu](mailto:tyamada@bu.edu)) Graduate Electrical & Computer Engineering

Mikhail Andreev ([mikh@bu.edu](mailto: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

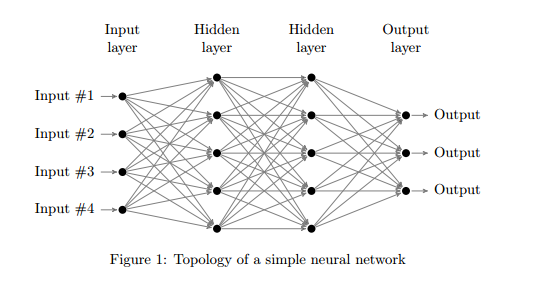
Character Recognition

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.

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.



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:

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

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.

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.

Base Case

Describe the base case – feed forward and back propagation.

Single Core Optimizations

Loop unrolling and multiple accumulators

Multiple Core Optimizations

Openmp and pthreads

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

Future Work

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 do improve as the number of neurons and layers increased.

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 self-supervised 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

Code Description