Fast Neural Network Training on General Purpose Computers

Harshit Kharbanda
University Of Illinois at Urbana-Champaign
kharban2@illinois.edu
*Masters-CS student

Roy H. Campbell
University of Illinois at Urbana-Champaign
rhc@illinois.edu

***Abstract*—Neural networks allow the implementation of complicated applications such as stock market predictions on low-end PCs. However, the training of neural networks can take many hours on a PC. In this paper we propose a technique for training complicated neural networks on a commodity GPU (available in a low-end PC) that completes 6 times faster than training on a multi core. Using the Proben1 benchmark for our analysis we use 15 datasets from 12 different domains to explore our solution. Our technique allows the training to be done with minimal CPU utilization time. This allows the user to carry out other tasks while the training is in progress. We compare several avenues of neural network training on a general purpose computer. The benchmark we use, covers problems of pattern classification from real life and hence is best suited for our tests as we aim to solve the problem of stock market predictions.**

***Index Terms*—Backpropagation, GPGPU, High-performance Computing, CPU Utilization, Stock Market Prediction**

I. Introduction

Artificial neural networks(ANNs) have been used for a variety of applications including pattern matching, data mining and prediction. Despite wide applications, ANNs are used with reservation as they need to be trained in order to operate effectively. Training needs large training sets and much processing time.

An ANN is built from a collection of artificial neurons where each neuron consists of an activation function defined according to the application. There are several types of ANNs in the literature and can be classified into networks based on supervised and unsupervised learning. In the supervised method, the neural network is provided with the input data as well as the target output whereas in the case of unsupervised learning, only the input data is provided. ANNs can be further classified into feedback and feedforward networks. Neural network types and learning methods have been classified scheme by Lippmann [1987]. In this paper we analyse the performance of the backpropagation algorithm on various avenues present on a low-end PC. Backpropagation can be used to train the multilayer perceptron(MLP), which is a type of feedforward Neural Network [2]. This particular ANN can be used for stock market predictions with up-to 72% accuracy [20].

A multilayer perceptron has one input layer and one output layer with one or more hidden layers. For our study, we have selected an MLP with 3 layers, i.e. one hidden layer. We implemented the ANN on an Nvidia Quadro GPU and optimized the backpropagation algorithm to suit the GPU architecture. We focused on exploiting the extensive parallelism offered by GPUs to speed-up the training of this MLP. This research focused on low-end PCs with GPUs to provide an inexpensive approach to stock market predictions(which use complicated neural networks). We have compared the performance of our implementation with sequential and parallel implementations of the same on multi-cores. The metrics we have used for our analysis are MLP training time and CPU utilization.

This paper is organized in seven sections. Section 2 describes the related work. In section 3, we discuss the backpropagation algorithm and its representation in matrix form. In section 4 we give the pseudo code of our implementation. Section 5 gives the experimental setup and Section 6 discusses the results and their analysis. Section 7 concludes the paper along with giving future directions of work.

II. Related Work

One of the early works to analyze the performance of general-purpose applications on GPU was by Che et al [12]. They compared the GPU and CPU performances of computationally demanding applications belonging to different categories of the Berkeley dwarves [13]. In their approach Backpropagation has been classified as belonging to the group of unstructured grid problems. The multi-core implementation by the authors has been written in Open-MP. We chose P-threads (over Open-MP) to implement the backpropagation algorithm on the multi-cores because they gave us more control on the execution of the threads and the role of each thread in training the network. The authors have used the GPU to offload the complex multiplications and use the CUDPP library as a layer of abstraction for parallel reductions.

GPUs are programmed using techniques like OpenGL [3] and DirectX [4]. With the increasing use of GPUs in general purpose computation, Nvidia CUDA [5], AMD CTM [6], Brook [7] and Accelerator [8] were developed. These APIs enable the programmer to write programs having two components, the kernel code and the host code. The kernel code runs on the GPU wheras the host code runs on the CPU and controls the entire execution including assigning kernel code to the GPU and controls data transfer between GPU and main memory. A variety of applications have been parallelized using GPUs. A recent survey by Owens et.al [9] gives a detailed

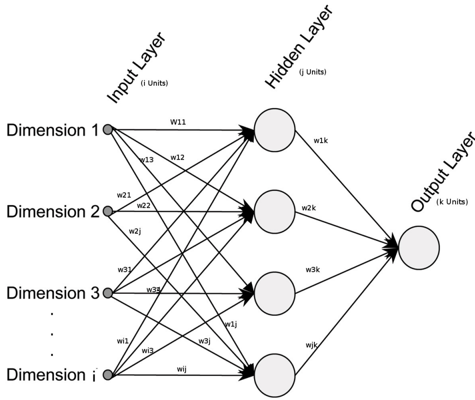

Fig. 1. Feedforward Neural Network

account of some of the more recent GPU applications and the speed-up achieved. In addition to single-GPU techniques, GPUs are currently being used in clusters for projects such as Folding@home [10] and Seti@home [11].

III. Neural Networks and the Backpropagation Algorithm

The biggest hurdle while using machine learning algorithms is the large learning time. Implementation of these algorithms on a GPU make them faster and thus shall help in increase of the use of neural networks and machine learning in solving pattern recognition problems on general-purpose laptops. We parallelized machine learning algorithms on a GPU in such a way that a single kernel function is executed on the various thread blocks of the GPU with different data-sets. In this section we discuss the parallel form of the backpropagation algorithm.

A Feedforward Neural Network is given in figure 1. The weights in this MLP are updated using backpropagation with gradient-descent technique. Training is achieved by minimizing the error criteria E(difference between derived outputs and actual outputs). The algorithm can be briefly explained as follows- To keep things simple, we have used a neural network with three layers. These are input, hidden, and output layers. The backpropagation algorithm can be divided into two stages, i.e., the forward pass and the backward pass. During the forward pass, the training data is supplied to the input layer, from where it propagates to the output layer through the hidden layer. Each node in the hidden layer gets inputs from all the nodes in the input layer. These inputs get multiplied by suitable weights before being added. The output of a hidden node is the non-linear transformation of this resulting sum. The output layer works in a similar fashion to calculate the output values. These output values are compared with target output values and the error is determined. This error is backpropagated to the hidden layer during the backward pass. During this phase, the connection weights between the layers are updated using the backpropagated error.

In the batch-mode variant of backpropagation, the mean square error is determined for each input pattern. This error is used to find the gradients of the hidden and final layers. The Δw for hidden and final layers are then calculated according to

$$\Delta w_{ij} = -\gamma * o_i * \delta_j$$

Where δ is the gradient for the hidden or the final layer. This process is repeated for all the input patterns and the cumulative Δw is determined. This cumulative Δw is then used to adjust the weights of the hidden and final layers. This whole process accounts for one epoch. The epochs are repeated until the mean square error becomes lower than the specified threshold value. In the batch-mode variant the descent is based on the gradient ∇E for the total training set.

$$\Delta w_{ij}(n) = -\epsilon * \frac{\partial E_p}{\partial w_{ij}} + \alpha * \Delta w_{ij}(n-1)$$

where ϵ and α are two non-negative parameters called learning rate and momentum. The momentum speeds up training in very flat regions of the error surface. It is necessary to propagate the whole training set through the network for calculating ∇E. This is one of the disadvantages of backpropagation as it can lead to very low training rates for large training sets. Backpropagation has been improved by having local learning rates ϵ_{ij} for every connection instead of a global value and also by having local backtracking.

Backpropagation can be implemented on the GPU in several ways depending on the specific technique used for updating the weights. The most suitable form for data parallel implementations is the batch gradient-descent. Here the network i.e. the current weights are fed to each of the GPU threads. The number of instantiated threads depends on the number of training sets.

IV. Implementation

We implemented sequential and parallel versions of backpropagation with gradient descent on multi-cores and GPUs. In all the implementations the weights were initialized to random numbers. For our tests we used the Dell Precision M4500 laptop which has Intel i7 processor with 8 cores and the Nvidia Quadro FX1800 GPU.

In the sequential implementation each input pattern is provided iteratively and the weights are updated. This process is repeated over the total number of training samples. We carried out the parallel implementation using P-threads [15]. In this implementation the total number of threads created is equal to the number of input patterns. Each thread takes an input pattern and trains the network to obtain the matrix of Δw_{ij}s. This training takes place in parallel and scales to utilize all the available cores. We calculate the cumulative Δw by adding all the Δw_{ij} matrices generated. This portion of the program is synchronized using a mutex variable. The threads repeat this training until the error value falls below the specified threshold.

We tried to optimize this parallel version of the code using the GSL-BLAS library [16]. The program written using the library performed better than the simple P-thread version when the number of epochs required to train the network was in the

Algorithm 1 Pseudo Code for GPU implementation of Backpropagation

```
//Initializing weight vectors
for i ← 0 to n-1 do
   for j ← 0 to k-1 do
      W1h[i][j] ← srand()
   end for
end for

for i ← 0 to k-1 do
   for j ← 0 to m-1 do
      W2h[i][j] ← srand()
   end for
end for

//Copy W1,W2,inp-h,target-h to device
W1d ← W1h
W2d ← W2h
target-d ← target-h
inp-d ← inp-h

//Define the kernel function
threads-per-block ← 512
blocks-per-grid ← ceil(p/512) //p represents the total number of training samples
thread-id ← blockId.x*blockDim.x+threadIdx.x

//The following code will be executed by all threads simultaneously. We have shown the execution of a single thread-id

//First layer output
O1[id] ← multiply(inp-d[id],W1)
O1[id] ← Activation(O1[id])
D1[id] ← O1[id] * (1-O1[id])

//Second layer output
O2[id] ← multiply(O1,W2)
O2[id] ← Activation(O2[id])
D2[id] ← O2[id] * (1-O2[id])

//Error and gradient calculation
E[id] ← target[id]-O2[id]
d2-d ← D2[id]*E[id]
d1-d ← D1[id]*W2*d2-d

//Transferring gradients to the host
d2-h ← d2-d
d1-h ← d1-d

//Calculating the Δws
Δw1 ← -r*d1-h*inp-h
Δw2 ← -r*d2-h*O1

//Weights are updated and the network is trained
```

order of a few hundreds. We also implemented the algorithm using the map-reduce [17] model on multi-cores. We used Phoenix [18] which is the state of the art map-reduce model provided for multi-cores. Phoenix is written using P-threads and scales well to utilize all the cores available. In the map phase the training was performed and the Δw_{ij} matrices were emitted as the intermediate key-value pairs. In the reduce phase these intermediate key-value pairs were added to obtain one final Δw. This reduce operation was performed by one reduce worker. This process accounts for one epoch and is repeated until the mean square error value falls below the specified threshold value. This version of the implementation did not perform well because of the overhead incurred in the map and reduce functions. Due to the poor performance of the training using Map-Reduce on multicores, we did not implement Backpropagation using Map-Reduce on GPUs. [21]

The GPU does not have a huge amount of memory. Due to this, the shared memory model which we used to implement backpropagation on multi-cores could not be used for the GPU implementation. The backpropagation algorithm given in [1] was edited so as to reduce the memory consumption. The altered algorithm used 50% lesser memory. The initial weights and the input and target vectors were stored in matrices. These matrices were copied to the GPU memory before beginning the training. Memory consumption was reduced by converting the δ_i matrices into single dimension matrices. The computations in the algorithm had to be accordingly altered. For each input vector a GPU thread was instantiated. These threads calculated their respective Δw_{ij} and stored the partial results in GPU memory. The GPUs are well utilized using this approach and are capable of handling thousands of threads at a time. Hence an application having thousands of input training vectors (like stock market prediction) scales well on the GPUs. This whole implementation was carried out by one kernel function. Hence the role of the CPU in the entire training was almost nil. After all the threads have determined their respective Δw_{ij} matrices, another kernel function is called to find the cumulative Δw matrix by adding all the Δw_{ij}. The execution of this kernel function was inspired by the divide and conquer technique. At any time during the execution of this kernel, the number of instantiated threads is n/2 where n is the number of Δw_{ij} stored in the GPU memory. This reduces the computation time compared to the case where addition would have been carried out by the host. It also saves the time taken to transfer the contents of GPU memory to the CPU.

V. Experimental SetUp

For evaluation purposes we used the Dell Precision M4500 mobile workstation with an Intel Core i7 Processor having 8 cores and an Nvidia Quadro FX 1800 GPU with 1GB memory. The GPU implementation was evaluated on the Dell Precision M4500 running Fedora 8 OS with CUDA 3.1. We used the Proben1 benchmark to evaluate our implementations. Backpropagation performed better on multi-core and GPUs than on multiprocessors because communication delays are lesser in these architectures. Between multi-core and GPU,

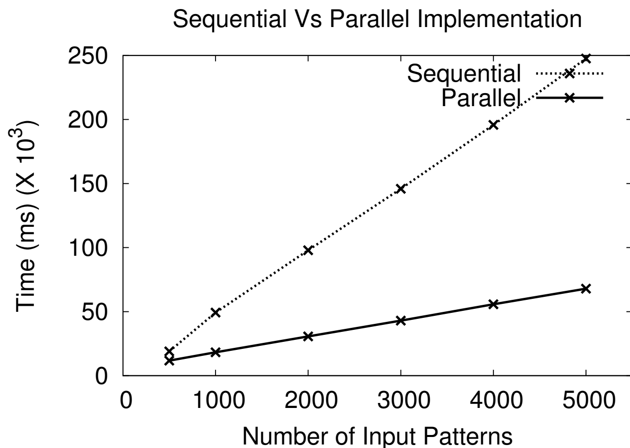

Fig. 2. Comparison of the running times of Backpropagation - sequential vs P-thread versions

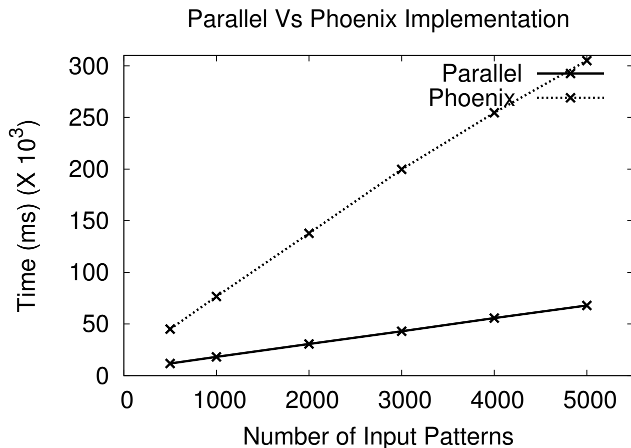

Fig. 3. Comparison of the running times of Backpropagation - P-thread version vs Phoenix implementation

the more the parallelism, better the performance of the GPU. This depends on the size of the data-set. The detailed analysis can be seen in the next section.

VI. Results and Discussion

Our first performance analyses sequential and parallel implementations of backpropagation on the CPU. In the parallel implementation, we used P-threads for determining the Δw_{ij} and the cumulative Δw was calculated in a sequential fashion using a mutex variable. The results obtained can be seen in the graph in figure 2. It can be seen that the parallel implementation of backpropagation using P-threads is much faster than the sequential implementation. As the number of input patterns increase, the sequential implementation becomes slower in comparison. For input sizes of 5000 patterns, the parallel implementation is 5 times faster than its sequential counterpart. The parallel implementation also scales well, utilizing 40 percent of the available cores. The sequential implementation on the other hand puts the entire load on a single core making the training slower.

We then compared the simple P-thread implementation with an equivalent implementation on Phoenix, the map-reduce framework for multi-cores. Phoenix was much slower for an input size of 5000 patterns. This is because the backpropagation algorithm does not fit the map-reduce model well. The overhead of splitting data among the map workers and then scheduling the workers is too much and does not result in considerable speedup. Also, only one reduce worker can do the job of combining all the outputs from the map workers. Scheduling more reduce workers using Phoenix makes the entire implementation slow. Overall no speedup(in comparison to the simple P-thread implementation) could be achieved when the network was trained using Phoenix as seen in Figure 3.

Our next comparison was between a parallel implementation on multi-cores using the GSL-BLAS library and a similar implementation using CUBLAS library [19]. The parallel CUBLAS implementation utilizes the GPU for carrying out matrix multiplications. The rest of the program runs on the CPU. The results obtained can be seen in the graph in figure 4. The GSL-BLAS implementation was found to be several times faster than the CUBLAS implementation. A peculiar CPU behavior was also seen when the parallel implementation using CUBLAS was used to train the network. Valleys can be observed in the CPU utilization graph in Figure 5 for CUBLAS implementation. The reason for this being that only one kernel can be active on the GPU at a time. The functions for matrix multiplication provided by the CUBLAS library are optimized kernels which run on the GPUs. In the parallel implementation all the threads are launched at the same time, hence they reach the matrix multiplication part of the program at roughly the same time. At these points the parallelism is effectively lost because the threads have to wait until the GPU is free. This is the reason for the valleys which can be seen in the CPU utilization graph. Though the time taken to carry out these matrix multiplications is low, it has a huge affect on the parallelism if the number of threads in operation is huge. Hence if the number of input patterns increase, the performance of the CUBLAS implementation decreases. Overall the GSL-BLAS implementation of the Backpropagation algorithm was around 1.5 times faster than its CUBLAS counterpart.

The fastest implementation of backpropagation on CPUs was found to be the parallel implementation using the GSL-BLAS library. We implemented the entire algorithm on the GPU instead of using it only for matrix multiplications as in the CUBLAS version. Our GPU version of backpropagation was found to be 6 times faster than the GSL-BLAS implementation. This can be attributed to the fact that a GPU provides a lot of cores that allows faster training of a large number of patterns. Multiple threads are distributed and scheduled on each core. Each thread handles one pattern and hence if the number of training samples increase, the GPU implementation becomes much faster than the fastest parallel implementation on multi-cores. The CPU utilization is also fairly low when

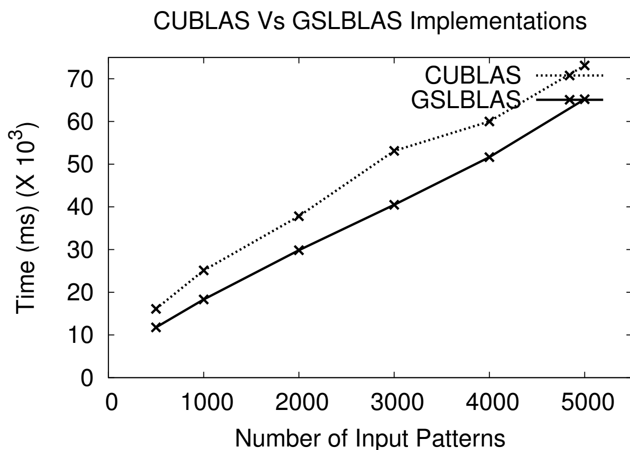

Fig. 4. Comparison of the running times of Backpropagation - parallel CUBLAS vs parallel GSL-BLAS

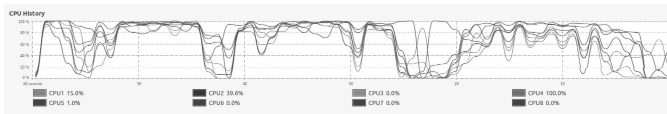

Fig. 5. CPU utilization of Parallel CUBLAS implementation

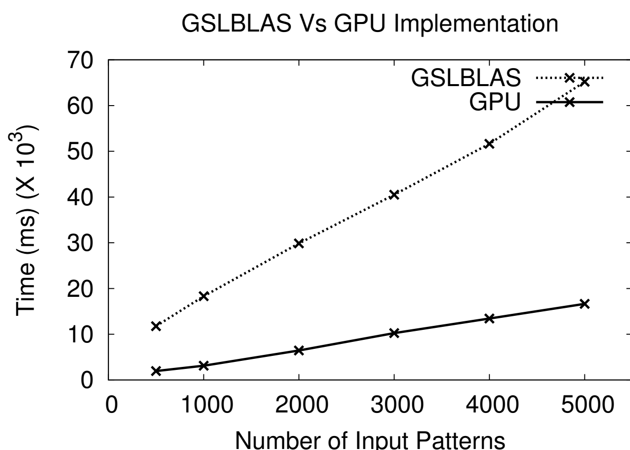

Fig. 6. Comparison of the running times of Backpropagation - parallel GSL-BLAS vs GPU implementation

the network is trained on the GPU. Most of the time only one core is 15 percent utilized which makes its utilization around 8 times lesser than that of the parallel implementation on multi-cores. These results have been plotted in the graph in figure 6. The training times depicted in the graph include the time taken to copy the contents from the host to the GPU.

VII. Conclusions and Future Work

This paper performs a detailed analysis of the training of feed forward networks on multi-cores and GPUs using back-propagation. We have chosen backpropagation due to its wide-spread applications especially in the areas of computational finance and pattern recognition. Our work concludes that the commodity GPU(available on a laptop) allows fast implementation of backpropagation on a low-end PC. Since GPUs are becoming ubiquitous computing units in modern day PCs, this encourages machine learning for common applications. The major hindrance in using neural networks is their large training time and required resources. GPUs can ameliorate this issue to a considerable extent. Our work suggests that the user of a PC can use the GPU while other applications proceed on the PC. This allows more effective utilization of the hardware at hand. We are currently working on improving prediction methods for computational finance using feed forward networks on GPUs.

References

[1] Raul Rojas. Neural Networks - A Systematic Introduction, *Springer*.1996.
[2] Leon Bottou Igor Durdanovic Hans Peter Graf, Eric Cosatto and Vladimire Vapnik. Parallel support vector machines: The cascade svm. In *NIPS,* 2004.
[3] OpenGL,http://www.opengl.org
[4] http://www.directx.org/
[5] NVIDIA CUDA(Compute Unified Device Architecture), http://developer.nvidia.com/object/cuda.html
[6] AMD CTM,http://ati.amd.com/products/streamprocessor/.
[7] Buck I, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston and P. Hanrahan. Brook for GPUs:Stream Computing on Graphics Hardware.*SIGGRAPH*.2004.
[8] Tarditi D. et al. Accelerator: using data parallelism to program GPUs for general-purpose uses.*ASPLOS*.2006.
[9] Owens J D et al. A survey of general purpose computation on graphics hardware.*Computer Graphics Forum*. Vol 26, 2007.
[10] Folding@home, http://www.scei.co.jp/folding
[11] SETI@home, http://setiathome.berkeley.edu
[12] Che et al. A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA. *Journal of Parallel and Distributed Computing, Elsevier*. 2009.
[13] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D.A.Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick. *The landscape of parallel computing research: A view from Berkeley, Tech. Rep. UCB/EECS-2006- 183, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley*. December 2006.
[14] Technical Brief: NVIDIA GeForce 8800 GPU Architecture Overview, November 2006
[15] Lewis B. and D.J.Berg.*Multithreaded Programming with Pthreads*.Prentice Hall,1998
[16] The GSL-BLAS library, http://www.gnu.org/software/gsl/
[17] Dean J and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Operating Systems Design and Implementation,* pages 137149, 2004.
[18] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis.Evaluating MapReduce for Multi-core and Multiprocessor Systems.*Proceedings of the 13th Intl. Symposium on High-Performance Computer Architecture (HPCA)*Phoenix, AZ, February 2007
[19] CUDA CUBLAS Library, NVIDIA, March 2008
[20] P.B. Patel, T. Marwala. Forecasting closing price indices using neural networks *IEEE Conference on Systems, Man, and Cybernetics* . 2006.
[21] Bingsheng He et al. Mars:a MapReduce framework on graphics processors.*Proceedings of the 17th international conference on Parallel architectures and compilation techniques.* pages 260-269,2008.