GNeuron: Parallel Neural Networks with GPU

Raghavendra D Prabhu
Computer Engineering, National Institute of Technology, Karnataka (NITK)
Karnataka, India
raghu.prabhu13@gmail.com

Abstract*— *Artificial Neural networks have been part of an attempt to emulate the learning curve of the human nervous system. However the vital difference of, nervous system being highly parallel and computer processor units remaining largely sequential persists. Here an attempt is made to bridge that gap with the help of Graphics Processing Units (GPUs) which are designed to be highly parallel. In particular Back propagation networks are considered which use supervised learning. Back-propagation algorithms, with no data dependencies are embarrassingly parallel and hence only a totally parallel system can exploit it fully. However, it has also been observed that GPUs underperform when either significant overhead in calculations is incurred or algorithm is not sufficiently parallel.

Index Terms— feed forward neural networks, Graphics processing units, Back-propagation networks, data-parallelism

I. INTRODUCTION

THE field of artificial neural networks has evolved over the past few years. Various neural network models have been proposed. Both supervised and unsupervised models of learning have been used. Back propagation is one such model which uses supervised learning for training. It uses an input layer for input perception, one or more hidden layers for appropriate functional manipulation of inputs and an output layer which decides the required pattern. The striking aspect of this model is that there are no feedback connections which reduce its complexity considerably. These attempts towards advanced learning capabilities have hit a roadblock with limited computation abilities compared to a human brain which has a tremendous parallel architecture, to provide a real time response.

Traditional Sequential computing systems which have been in use for quite long time are all ill-equipped to handle the humungous processing capabilities neural systems want. Even though attempts have been made by adding processor cores, software and hardware threading; true parallelism has still remained elusive.

Graphics Processing Units (GPUs) have been used for the purpose of high quality graphics rendering for quite some time. However, it has been hardly used otherwise. Nowadays all the medium-to-large computers come with relatively inexpensive GPUs. The enormous parallel infrastructure of GPU is therefore wasted. The Central Processing Unit (CPU) therefore gets bottlenecked with all the computational tasks. Also the Moore's law [9] for silicon is nearing saturation. Adding more cores or multiprocessors is not helping either. The GPUs on the other hand have witnessed rapid growth [10] over the past few years with few cores in the beginning to 128 cores now.

This work therefore attempts to provide a parallel infrastructure for the above mentioned neural computation model through Stream processing [5] with help of GPUs. This way the neural engines can be truly over-clocked to be of importance in real-life situations. The presented

model implemented over Microsoft Accelerator [1] C# Platform provides a 2x to 10x improvement over its slower counterpart.

The remainder of this paper is organized as follows: Section II contains related work, Description of problem domain is done in Section III, Section IV describes the design of solution with parallelization and Section V provides experimental results with appropriate comparisons.

II. RELATED WORK

Most of the work in optimizing the neural networks has been mostly at algorithmic level (tweaking the algorithm itself). [6] is one such which achieves speedup by adapting a different training methodology like parallel dynamic learning concept. The primary advantage of this method is that it is independent of any architecture constraints. However, it is also true that poor implementation can diminish the speedup achieved. Making synaptic weights a function of certain variables [7] is done to improve sensitivity of the network. There are some implementations [11] which propose parallel implementation using threads, but as explained before threads bring with them their own problems of synchronization and management overhead. In the same way [2] proposes PVM (Parallel Virtual Machine), Pthreads, MPI for Self Organizing Maps (SOM) but they suffer from their own limitations. Using clusters [3] suffers from communication overheads and contention delays. One striking aspect of these approaches is that, if the algorithm gets more and more complex (i.e. more sequential), parallelizing it becomes more difficult and consequently the GPU implementation becomes more difficult and less fruitful than its counterparts which can handle that to some extent.

III. PROBLEM DESCRIPTION

A. Overview

The Back-propagation network considered here is the simplest of all in its kind with no optimizations at the algorithmic level. Hence, with no adjustments the implementation becomes purely sequential. Scaling it further makes the situation worse with an exponential complexity. Therefore, the primary task here is to optimize at an external level – at SPMD (Single Program Multiple Data) level.

Parallelizing it can be done in many ways. Firstly, a pseudo-parallel approach can be adopted where parallelism can be exploited through a threading mechanism. However, the net gain obtained is less after accounting the thread overhead in their maintenance and synchronization issues. Moreover, it is achieved at the cost of several pipeline and other delays burdening the CPU. Other avenues like OpenMP, MPI (Message Passing Interface) also exist, but they again add further to the problem of delaying the CPU.

Problems like that of Back-propagation networks are “massively parallel” where distribution is to the fullest extent waiting to be exploited. Also very little control like branching structures are required here. This serves as an ideal candidate for GPU since the fragment processors operate as a grid on the independent units of data array. The next section will detail the design of the parallel algorithm.

IV. DESIGN OF THE SOLUTION

A. Sequential Design

Design of the solution is done considering a sequential design first and then identifying the problematic areas. The primary focus of speedup is the training cycle of the Back-propagation feed-forward network. I chose the training cycle because a well trained network can respond to inputs in testing phase in real-time. Also, in systems where data is dynamic -changing continuously with time-training it on new data takes its toll on the efficiency of the system if incremental training is unavailable.

The training is separated into following stages:

1. Initializing the input and output vectors
2. Initializing the weights to random values and normalizing them.
3. Repeating the following for number of steps equal to number of cycles or until error is below threshold:
   - 3.1 Repeat the following for each pattern:
     - 3.1.1 Calculate the hidden layer output.
     - 3.1.2 Based on this calculate the output layer output.
     - 3.1.3 Compare the two outputs to get error vectors.

3.1.4 Back-propagate the errors by updating the weights between input and hidden layer and hidden layer and output layer.

3.1.5 Update the learning constant 'beta' and threshold 'theta' of all neurons.

B. Design with Parallelization

The design of the algorithm stresses on identifying the potential "hotspots" of computation in the algorithm and reducing their complexity. However there are certain places in the algorithm which are better left non-optimized since the cost involved in doing so exceeds the net gain obtained. The optimizations have also taken into consideration the platform on which the program is developed.

Parallelization is not done in the following sections:

1. Initializing the input, output and the weights. Since, initializing does not involve any expensive computation.
2. Normalizing the weights since it is not repetitive and the sequential version has lesser overhead.
3. Final and the most important one is that training for a number of cycles is done sequentially. The prime reason for this being the constraints imposed as a result of hardware chosen. The current GPU platforms [1] support only arrays of dimension two. Since two dimensions are already required to store multiple patterns, cycles cannot be parallelized.

However the above implications do not impose a significant impediment on the performance of the application as gain obtained through the parallel sections overshadows this.

Before parallelizing, the data structures need to be redesigned. The Accelerator library provides two closely related structures:

1. Floating Parallel Array (FPA).
2. Disposable Floating Parallel Array (DFPA).

The first one is either freshly initialized or obtained as result of parallel operations. But the latter one can be used to create parallel arrays out of existing "primitive" arrays but need to be explicitly "disposed". Hence the latter one is used in beginning and in latter portions of program FPAs are used.

The following sections of the algorithm have been extensively parallelized:

1. Four DFPAs are created from existing input, output and the two weight vectors (weight1 and weight2).The dimensions of the vectors are as in Table. I

Table I

Input	number of patterns × number of neurons in input layer
weight1(between input and hidden layer)	number of input layer neurons × number of hidden layer neurons
weight2(between hidden and output layer)	number of hidden layer neurons × number of output layer neurons
Output	number of patterns × number of neurons in output layer

2. The following steps are repeated until error is lower than threshold or number of cycles is exhausted:

(Note: Steps detailed here forth use the primitives (prefixed with 'PA.' which stands for Parallel Array) provided by the GPU to implement vectors additions, multiplications. Only important steps are shown.)

2.1 The inner product of input and weight1 is obtained and an activation function is applied to obtain the hidden layer activation as shown in Table II.

Table II

```
FPA temp =
PA.Add(PA.InnerProduct(input,
weight1),theta);
//PA.InnerProduct == Matrix
//multiplcation
FPA hidden_activation =
PA.Reciprocal(PA.Add(PA.Pow(new
FPA(2.71828f,new
int[]{num_patterns,number_hidde
n}),PA.Negate(temp)), 1.0f));
```

Comparing the Table II and Table III it is evident that two GPU instructions have replaced three nested loops of expensive calculations. Therefore, two instructions

execute in 'unit' time on the GPU compared to that on CPU.

Proceeding in the same direction other loops in the computation can also be parallelized as follows:

Table III

```
for (int k = 0; k < numpat;
k++)
{
 for (int i = 0; i < nh; i++)
 {
  hidden[k, i] = 0.0f;
  for (int j = 0; j < ni;j++)
  {
   hidden[k, i] +=
   iwt[j,i]* input[k,j];
  }
  hidden[k, i] += theta[k,i];
  hidden[k, i] =
1.0f/(float)(1.0+Math.Exp(-
hidden[k, i]));
 }}
```

2.2 Obtaining the output of output layer from hidden layer activations and weight2 and applying another activation function.

2.3 Calculating the error in the output as in Table IV.

Table IV

```
FPA error =
PA.Subtract(desired_output,
actual_output);
```

2.4 Correcting the errors in the weight vectors based on error in the output and updating them as shown in Table V.

Table V

```
FPA output_weight =
PA.Add(PA.Multiply(PA.InnerProd
uct(PA.Transpose(hidden,new
int[]{1,0}),error),output_beta)
,output_weight);

FPA input_weight =
PA.Add(PA.Multiply(PA.InnerProd
uct(PA.Transpose(input,new
int[]{1,0}),hidden_error),
hidden_beta),input_weight);
```

2.5 Final and the most important step being converting the disposable parallel arrays into normal 2-D arrays as shown in Table VI.

Table VI

```
PA.ToArray(output_weight,out
array_output_weight);
```

The important thing to note here is that all the computations performed so far are delayed until the last step of converting them into primitive arrays and is a case of 'delayed computation'.

The primary advantage of this is decreased communication overhead with GPU and thus maximum performance. However, computation is enforced in some places by the use of 'Eval' function.

Table VII

```
FPA oerror =
PA.Eval(PA.Subtract(actual_outp
ut, correct_output));
```

This forces computation to be performed on data. The placement of these needs to be done intelligently to ensure maximum performance so that computations queued up does not exceed the capacity or memory of GPU. This is decided on basis of trial-and-error basis.

V. EXPERIMENTAL OBSERVATIONS

A. Environment

The following experiments are conducted on a personal computer running AMD Turion X2 1.66 GHz and Nvidia GeForce 6150 Go GPU with 256MB Video RAM and Pixel Shader Version 3.0. Hence the maximum array size is 4096 × 4096.The Operating System chosen is Microsoft Windows XP Service Pack 2 with DirectX 9.0c and runtime environment is Microsoft .NET 2.0 with C# as the language. Microsoft Research Accelerator [1] GPGPU library is used.

B. Results

An object-oriented application is written containing the code mentioned in design and is executed in the platform mentioned above.

Two major parameters are varied in the evaluation of application:

1. Number of patterns: This parameter affects the dimensionality of input and output

vectors. This is also equal to number of distinct patterns presented in each cycle.

2. Network size: This is equal to number of input neurons × number of hidden layer neurons × number of output neurons. This is typically of order 10^6.

These parameters are varied against time in seconds. Accurate measurement of time is accomplished using DirectX Timer.

The following observations can be made:

1. As can be noted in Fig.1 initially when the number of patterns is less(here for 100) the CPU time and GPU are almost equivalent and for still lesser values CPU dominates but as number increases growth rate of CPU curve is nearly exponential compared to GPU. This is due to the fact that at lesser values overhead dominates the gain obtained through parallelization and vice-versa.
2. Fig.2 presents another instance of the growth mentioned above. For values in range 0-200, CPU dominates. But as the network is scaled up , CPU time increases at a tremendous rate whereas GPU time series grows relatively slowly.

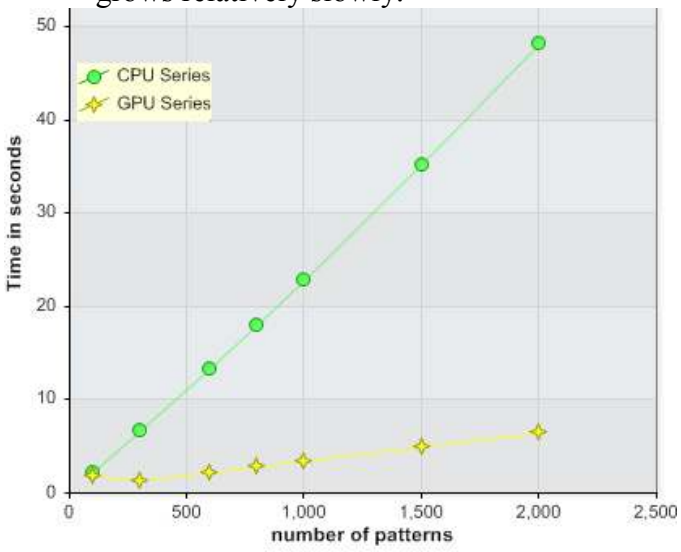

Figure 1 shows time variance of the application with the number of patterns

REFERENCES

[1] David Tarditi, Sidd Puri, and Jose Oglesby.:Accelerator: using data-parallelism to program GPUs for general-purpose uses.Technical Report MSR-TR-2004-184, Microsoft Corporation, December, 2005.

[2] Udo Seiffert .: Artificial Neural Networks on Massively Parallel Computer Hardware. ESANN 2002: 319-330.

[3] Hannes Schabauer, Erich Schikuta, Thomas Weishäupl: Solving Very Large Traveling Salesman Problems by SOM Parallelization on Cluster Architectures. PDCAT 2005: 954-958.

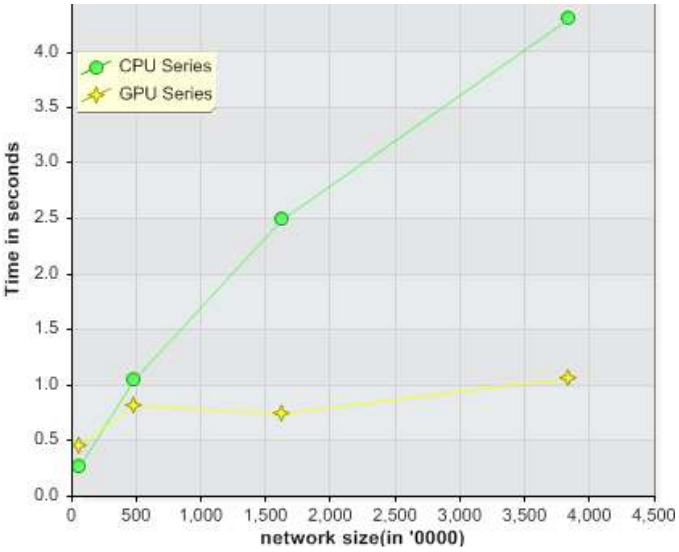

Figure 2 plots time in seconds against the network size

[4] David P. Luebke, Mark Harris, Naga K. Govindaraju, Aaron E. Lefohn, Mike Houston, John D. Owens, Mark Segal, Matthew Papakipos, Ian Buck: S07 –“ GPGPU: general-purpose computation on graphics hardware”. SC 2006: 208.

[5] Suresh Venkatasubramanian.:The Graphics Card As A Stream Computer.SIGMOD-DIMACS Workshop on Management and Processing of Data Streams.

[6] Lursinsap, C.; Kim, J.H.: Parallel learning for back-propagation network in binary field. Circuits and Systems, 1991., IEEE International Sympoisum on Volume , Issue , 11-14 Jun 1991 Page(s):1477 - 1480 vol.3.

[7] Obayashi, M. Kobayashi, K.: A new method for faster neural networks learning introducing functions of synaptic weights.Neural Information Processing, 1999. Proceedings. ICONIP '99. 6th International Conference on Publication Date: 1999 Volume: 3, On page(s): 1178-1183 vol.3.

[8] Alexandra Cristea and Toshio Okamoto.: A Parallelization Method for Neural Networks with Weak Connection Design. Proc. of the International Symposium on High Performance Computing,pp. 397-404, 1997.

[9] Gordon E. Moore.: Cramming more components onto integrated circuits. Electronics, Volume 38, Number 8, April 19, 1965.

[10] David B. Kirk.: The Future of Graphics Computing.

[11] Thulasiram, R.K. Rahman, R.M. Thulasiraman, P. Neural network training algorithms on parallel architectures for finance applications. International Conference on Parallel Processing Workshops, 2003. Proceedings. 2003 Publication Date: 6-9 Oct. 2003. On page(s): 236- 243.