Parallel Training of a Back-Propagation Neural Network using CUDA

Xavier Sierra-Canto
División Industrial
Universidad Tecnológica Metropolitana
Mérida, México
xavier.sierra@utmetropolitana.edu.mx

Francisco Madera-Ramírez
Facultad de Matemáticas
Universidad Autónoma de Yucatán
Mérida, México
mramirez@uady.mx

Víctor Uc-Cetina
Facultad de Matemáticas
Universidad Autónoma de Yucatán
Mérida, México
uccetina@uady.mx

***Abstract*—The Artificial Neural Networks (ANN) training represents a time-consuming process in machine learning systems. In this work we provide an implementation of the back-propagation algorithm on CUDA, a parallel computing architecture developed by NVIDIA. Using CUBLAS, a CUDA implementation of the Basic Linear Algebra Subprograms library (BLAS), the process is simplified; however, the use of kernels was necessary since CUBLAS does not have all the required operations. The implementation was tested with two standard benchmark data sets and the results show that the parallel training algorithm runs 63 times faster than its sequential version.**

***Keywords*-neural networks; back-propagation; CUDA**

I. Introduction

Training an artificial neural network is time-consuming due to the large number of epochs and weight updates required to reach an optimal performance. Several efforts have been made to increase the convergence speed or to reduce the computational cost [6], [18], [19], [3].

The use of a Graphics Processing Unit (GPU) is an alternative for faster execution of general purpose operations [8], [11]. Carpenter presents cuSVM [4], a software package for high-speed Support Vector Machine (SVM) training and prediction that exploits the massively parallel processing power of Graphics Processors (GPUs). cuSVM is written in NVIDIAs CUDA C-language GPU programming environment, includes implementations of both classification and regression, and performs SVM training (prediction) at 13-73 (22-172) times the rate of state of the art CPU software. Another relevant work is the one presented in [5], where the authors explore the effectiveness of GPUs for a variety of application types and describe some specific coding idioms that improve their performance on the GPU. The authors also discuss advantages and inefficiencies of the CUDA programming model and some desirable features that might allow for greater ease of use and also more readily support a larger body of applications.

Furthermore, GPUs have been already employed in the training of neural networks using different algorithms and programming languages [20], [14], [15]. In [21] the authors propose a new neural network model which they call Locally-connected Neural Pyramid (LCNP). This model is optimized for large-scale, high-performance object recognition. LCNP involves the use of a hierarchical structure of two-dimensional maps. The main features of their model are its hierarchical structure, the local connectivity without weight sharing, the use of sub-sampled inputs in all hierarchical layers, and the local preprocessing of the input patterns. This implementation is up to 82 times faster than a single-core CPU version of the system. It was tried with 3 databases: MNIST, NORB and LabelMe, using a Nvidia GeForce GTX 285 card.

The simulation of spiking neural networks using a GPU-SNN model, running on an NVIDIA GTX-280 with 1GB of memory is presented in [13]. Results shows that such a model is up to 26 times faster than a CPU version for the simulation of 100K neurons with 50 Million synaptic connections, firing at an average rate of 7Hz. Similar results were obtained by Bernhard *et al.* [1].

The Restricted Boltzmann Machine (RBM), a popular type of neural network is tested on a NVIDIA GTX280 GPU, resulting in a computational speed of 672 million connections-per-second and a speed-up of 66-fold over an optimized C++ program running on a 2.83GHz Intel processor [12].

Since NVIDIA presented CUDA, new options for programming neural networks have been available for designers. Moreover, CUBLAS brings an easy way for programming several steps of the back-propagation algorithm. In this paper, we propose the implementation of a back-propagation neural network using CUDA and CUBLAS. Such an implementation runs on NVIDIA GPUs using CUDA based on two stardard benchmark sets provided by PROBEN1 [16]. Promising results were obtained using a NVIDIA Tesla C1060 card and one hidden layer of neurons in the order of 63x compared to a typical CPU.

The rest of the paper is organized as follows. In Section II, we give an introduction to neural networks and the back-propagation algorithm. The CUDA environment is explained in Section III. In Section IV, we provide the details of our parallel implementation on CUDA. Section V presents our experimental work, and finally, in Section VI, we conclude with the benefits, drawbacks and our future work.

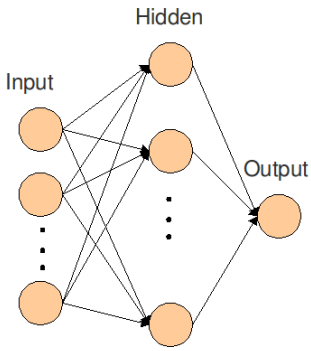

Figure 1. A fully connected neural network with one hidden layer and one output unit.

Algorithm 1 $(A, B\ )$

Forward Propagation

1. $\vec{a}_n\ = \vec{y}\ W_1$

2. $\vec{a}_n\ =\ f(\vec{a}_n)$

3. $out\ = \vec{a}_n\ W_2$

4. $out\ =\ f(out)$

Backward Propagation

5. $d_k\ =\ (t - out)\ f'(out)$

6. $\vec{d_h}\ =\ W_2\ d_k\ f'(\vec{a}_n)$

7. $W_2\ =\ W_2\ +\ \eta\ \vec{a}_n\ d_k$

8. $W_1\ =\ W_1\ +\ \eta\ \vec{y}\ d_h$

Figure 2. The sequential algorithm contains 8 steps, divided in two parts: forward and backward propagation.

II. The back-propagation algorithm

An artificial neural network is an information-processing system with certain performance characteristics in common with biological neural networks [2], [7], [9]. Artificial neural networks are based on the following assumptions:

- Information processing occurs at many simple elements called neurons.
- Signals are passed between neurons over connection links.
- Each connection link has an associated weight, which, in a typical neural net, multiplies the signal transmitted.
- Each neuron applies an activation function, usually non linear, to its net input to determine its ouput signal.

Since 1986, when Rumelhart and McClelland [17] presented the back-propagation algorithm, it has been used and adopted as a standard training method for neural networks. For a fully connected network (see Fig. 1) a matrix representation of the back-propagation algorithm consists of the following steps:

First, initialize the input layer including an input for bias,

$$\vec{y_0} = \vec{x},$$

then, propagate activity forward trough layer $l\ =\ 0$ to $l = L$,

$$\vec{y_l} = f(W_l y_{l-1}),$$

where W is the random weights matrix and f is the activation function.

Next, calculate the error in the output layer,

$$\vec{\delta_L} = (\vec{t} - \vec{y}_L)\ f'(\vec{y}_L),$$

after that, back-propagate the error trough layers,

$$\vec{\delta_l} = (W_{l+1}^T \vec{\delta}_{l+1})\ f'(\vec{y_l}),$$

and finally update the weights,

$$W_l \leftarrow W_l + \eta\ \vec{\delta_l}\ \vec{y}_{l-1}^T.$$

Figure 2 shows the back-propagation pseudocode implemented for a neural network with one hidden layer.

As we can see, several vector and matrix operations are made in this training algorithm so that the implementation in a parallel program is clearly suitable.

III. The CUDA environment

CUDA (Computed Unified Device Architecture) is the name of a general purpose parallel computing architecture of NVIDIA GPUs. A GPU is a data pipeline processor capable to execute a function in a set of input records, producing a set of output records in parallel. These functions and data are not necessarily pixels and shaders, but data such as mathematical equations.

The CUDA programming model is based on the concept of a kernel. A kernel is a function that is executed multiple times in parallel, each instance running in a separate thread. The threads are organized into one-, two-, or three-dimensional blocks which in turn are organized into one- or two-dimensional grids. The blocks are completely independent of each other and can be executed in any order. Threads wihtin a block however are guaranteed to be run on a single multiprocessor.

Hundred of kernels can run collectively thousands of threads. A kernel can share sources, including memory and records. Serial code should be run in the CPU, while parallel code should be run in the GPU.

Since data structures employed are the arrays, vectors and matrices, the CUBLAS library was utilized. CUBLAS is an implementation of BLAS (Basic Linear Algebra Subprograms) on the top of CUDA and is self-contained at the API level so there is no need of direct interaction with CUDA . It provides helper functions to create matrix and vector objects in GPU memory space, fill them with data, perform BLAS operations and read back the results.

CUDA has been previously used for parallel implementations of spiking neural networks [13]. It has also been tested and compared to implementations in computer clusters running OpenMP [10].

IV. Parallel Implementation

The parallel implementation of the back-propagation algorithm consists of matrix and vector operations performed on the GPU, where data are kept on the device as long as possible.

There are two types of parallel operations: vector-matrix products and arithmetic operations. In the first type of operations, where the CUBLAS library is utilized, a vector is seen as a unidimensional matrix of size $1 \times n$. In the second type of operation, where a kernel is launched, a function is applied over all the elements of a vector.

Focusing on the vector matrix product, in the first step, $\vec{a}_n$ of size $1 \times n$ is obtained by the product of a vector $\vec{y}(1 \times m)$ by a matrix $W_1(m \times n)$. In the third step, out is the result of $\vec{a}_n(1 \times n)$ by $W_2(n \times 1)$. Finally, in step 6, $W_2(n \times 1)$ is multiplied by $d_k(1 \times n)$.

The second type of operation is the application of a function over all the elements of a vector. This is achieved by looping over all the elements in the input array, computing the operations of such a function, and writing this change to the current element of the output array. In steps 2 and 4 a sigmoidal function is performed, whereas in steps 5 and 6 the derivative of the sigmoidal function is executed.

A. Using CUBLAS

Assuming a vector of length N and a matrix of order $N \times N$, then the number of floating points operations for its product is $O(N^2)$. The basic model by which applications use CUBLAS library is to create matrix and vector objects in GPU memory space, call the CUBLAS function, and, finally, move the results from GPU memory back to the host.

Fortunately, the blocks and the threads definition is not a problem due to CUBLAS provides a set of algorithms structured according to an optimum parallelism. Two functions were chosen: *cublasSgemm* and *cublasSdot* which perform the matrix-matrix product and the dot product respectively.

The *cublasSgemm* computes the matrix-matrix operation $C = \alpha * A * B + \beta * C$, where A, B and C are matrices and α and β are scalars. Setting $\alpha = 1$ and $\beta = 0$, the computation is reduced to $C = A * B$. This function serves to obtain steps 1 and 6 of Algorithm 1.

cublasSdot is utilised for step 3, given the result in $O(1)$ time. *cublasSdot* returns the dot product of the single-precision vectors if succesful, and $0.0f$ otherwise.

B. Using kernels

A kernel is a function callable from the host and executed on the CUDA device, simultaneously by many threads in parallel.

Fig. 3 shows the kernel Sigmoid, where col is the parallel step and all the threads are accessed in one time. $blockIdx$ is the block identifier, $blockDim.x$ is the block's size specified previously, and $threadIdx.x$ is the thread identifier. The final x in these values indicates an unidimensional arrangement; in case the final y is used, a bidimensional arrangement would be chosen.

Sigmoid (a_n)
1. $col = blockIdx.x * blockDim.x + threadIdx.x$
2. $a_n[col] = 2/(1 + e^{-2a_n[col]})$

Figure 3. The kernel sigmoid is called from the host with the instruction Sigmoid$<<< Grid, Block >>> (a_n)$.

C. The parallel algorithm

Having described the two kinds of parallel operations, the whole parallel algorithm is detailed straightforward. CUBLAS must be initialized and memory should be allocated. Rather than using *cudaMalloc*, *cublasAlloc* is employed; in fact *cublasAlloc* is a wrapper around *cudaMalloc*. Even when the number of blocks, $dimGrid$, and the number of threads per block, $dimBlock$, are not required by CUBLAS, they should be defined at the start of the code to be used when the kernels are launched. The forward propagation part is illustrated in Figure 4, where the first 4 steps of the algorithm are performed.

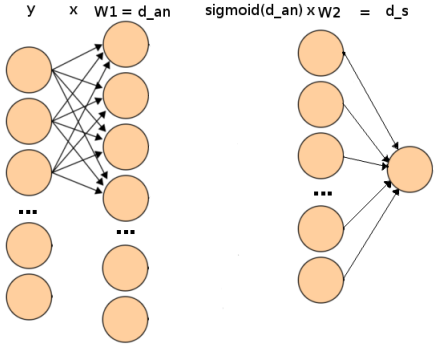

Figure 4. The forward propagation involves two products: $\vec{y} \times W_1$ and $sigmoid(d_{an}) \times W_2$, and two kernels: $sigmoid(d_{an})$ and $sigmoid(d_s)$.

Parallel Algorithm
1. *Allocate memory space for device with* cublasAlloc
2. *Set matrix and vector variables with* cublasSetVector, cublasSetMatrix
3. cublasSgemv*(W_1, $\vec{y}$)*
4. *sigmoid$<<< Grid, Block >>>$(an,1)*
5. cublasSgemv*(W_2, $\vec{an}$)*
6. *sigmoid$<<< Grid, Block >>>$(out,1)*
7. *Compute output error dk on CPU*
8. *error-h$<<< Grid, Block >>>$(an,W2,dk,1)*
9. *act-W2$<<< Grid, Block >>>$(an,W2,dk,1)*
10. cublasSgemm*(y,dh)*

Figure 5. The pseudocode of the parallel algorithm contains six CUBLAS operations and four kernel calls.

In step 1, hardware resources necessary for accessing the GPU are allocated with *cublasAlloc*. The matrices of the algorithm are as follows: the weights $W_1(m \times n)$, $W_2(n \times 1)$, the features matrix of size $(m \times k)$, and the input vector $\vec{y}$ of size n. In step 2, the data defined previously are copied from the CPU to the GPU. CUBLAS computes the optimum number of blocks and threads to perform its functions, which basically depends on m, n, and k. In the features matrix the column i of size m is copied to the GPU in the ith of h trainings.

In step 3, the matrix-vector product is utilized to propagate the data forward to the hidden layer, $\vec{y}(1 \times n)\ W_1(m \times n)$. The first kernel is launched in step 4 and is detailed in Fig. 2. The number of blocks was set to 240 since the Tesla card has the same number of multiprocessors with 8 scalar processors each. To balance the workload, the number of threads was chosen to be $240/n$. This kernel is called again in step 6 when the network is propagated to the output. This propagation is obtained by the matrix-vector product $\vec{y}(1 \times n)\ W_2(n \times 1)$ in step 5.

The number of threads should be as large as possible to enable the CUDA scheduler to better utilize the available computational power by executing threads when other ones are waiting for global memory transfers to be completed. The output error is performed on the CPU, $d_k += (target - d_n)^2$, in the $7th$ step.

The last two kernels are shown in Fig. 6 and 7 respectively. For each hidden neuron, the error is computed as stated in step 8. The weights are updated, using kernel *act* for W_2 and kernel *cublasSgemm* for W_1.

$\mathbf{error - h}\ (dh, d_{an}, W_2\ )$
1. $col\ =\ blockIdx.x * blockDim.x + threadIdx.x$
2. $dh[col]\ =\ 0.5 * (1 - d_{an}[col]) * (1 + d_{an}[col]) * W_2$

Figure 6. The kernel error is called from the host with the instruction error-h$<<<\ Grid, Block\ >>>\ (dh, d_{an}, W_2)$.

$\mathbf{act - W2}\ (d_{an}, d_k, W_2\ )$
1. $col\ =\ blockIdx.x * blockDim.x + threadIdx.x$
2. $W_2[col]\ +=\ d_{an}[col] * d_k * 0.001$

Figure 7. The kernel act is called from the host with the instruction act$<<<\ Grid, Block\ >>>\ (d_{an}, d_k, W_2)$.

V. Experimental work

We have tested our implementation with two standard sets provided by PROBEN1 [16]: the *cancer1* with a feature vector of 9 elements and the *mush1* with 125. For the former, 525 examples were used as the training set and 275 were reserved for the test. In the latter, the first 6093 examples were used for training and 2031 for testing. All the experiments were made without considering evaluation sets.

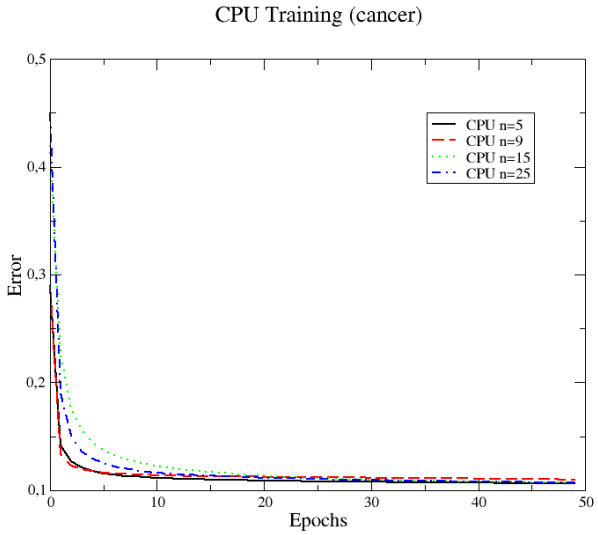

Figure 8. The number of epochs against the error using the *cancer1* dataset running on CPU. The curves were generated with different numbers of hidden neurons ranging from 5 to 25.

The activation function was the bipolar sigmoid (1) which provides an output from -1 to 1.

$$sig(x) = \frac{2}{1 + e^{-2x}} - 1 \qquad (1)$$

The learning rate was kept constant $\eta = 0.01$ and the weights were always randomly initialized in the interval $(-0.5, 0.5)$.

The NVIDIA card employed is the Tesla $C1060$ with 240 kernels, 3.7 Teraflops single precision, 312 Gigaflops double precision, and 4 GB of memory. This card is installed in a desktop Intel Core2 Duo E6750 @ 2.66 GHz and 4GB RAM.

A. Cancer data

Figures 8 and 9 show how the training error is reduced by incrementing the number of epochs as it is expected. Each curve represents the training process with a different number of hidden neurons, ranging from 5 to 25. The final classification error is 0.12766.

Figure 10 depicts a CPU-GPU training time comparison varying the number of hidden neurons from 1 to 25. A logarithmic time scale was regarded for illustration purpose. The GPU implementation is much faster than the CPU in 46x. As the number of hidden neurons increases, the sizes of the matrices W_1 and W_2 grow, and therefore the computation becomes more complex.

B. Mushroom data

Figures 11 and 12 show how the training error is reduced by incrementing the number of epochs as it is expected. Each curve represents the training process with a different

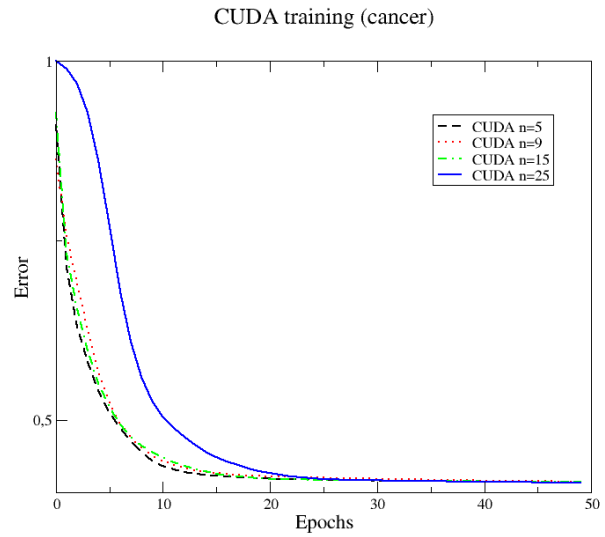

Figure 9. The number of epochs against the error using the *cancer1* dataset running on GPU. The curves were generated with different numbers of hidden neurons ranging from 5 to 25.

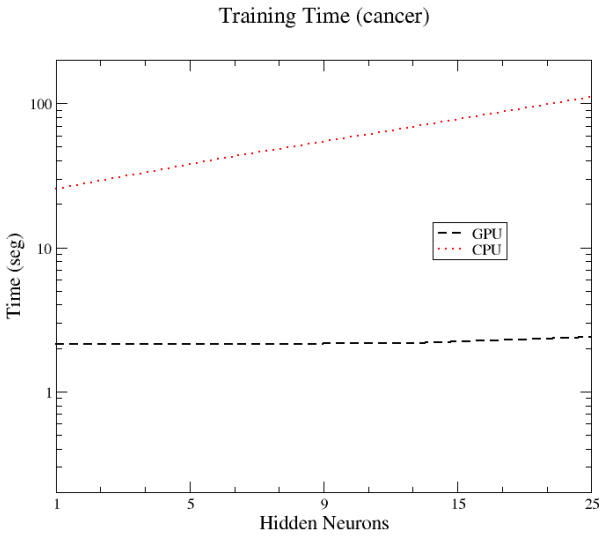

Figure 10. The number of hidden neurons vs the training time in seconds with the *cancer1* dataset. A logarithmic time scale is used for illustration purposes.

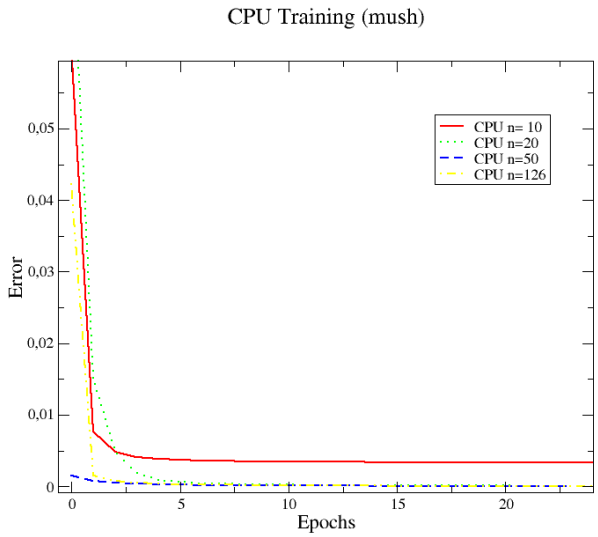

Figure 11. The number of epochs against the error using the *mush1* dataset running on CPU. The curves were generated with different numbers of hidden neurons ranging from 10 to 126.

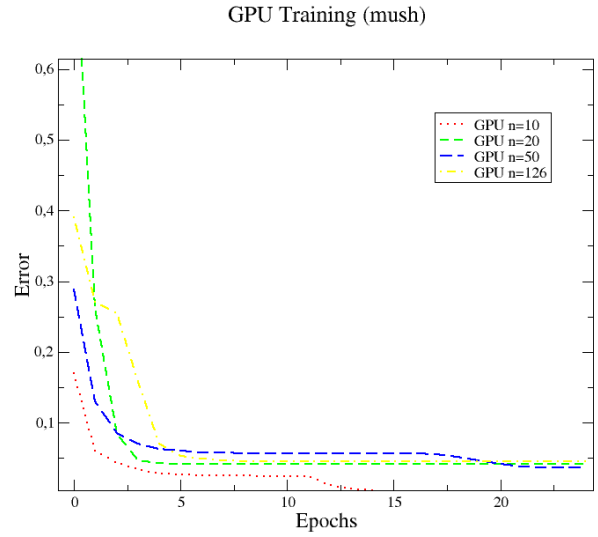

Figure 12. The number of epochs against the error using the *mush1* dataset running on GPU. The curves were generated with different numbers of hidden neurons ranging from 10 to 126.

number of hidden neurons, ranging from 10 to 126. The final classification error is 0.0004.

Figure 13 illustrates a CPU-GPU training time comparison varying the number of hidden neurons from 1 to 126. A logarithmic time scale was regarded for illustration purpose. As we can see, the GPU implementation is also much faster than the CPU in 63x. This improvement is larger than in the previous experiment due to the size of the feature vector which is also larger, being in the *mush1* dataset of 126, meanwhile in the *cancer1* is only 9. Aside from that, the number of hidden neurons were increased from 10 to 126 which also contributes to the final complexity. Given the algorithm parallel design, the workload is distributed among the threads looking always for the best balance. This implies that the CPU-GPU processing time difference gets larger as the number of operations increases.

VI. Conclusions

In this paper we have proposed a parallel implementation of the back-propagation algorithm for neural networks using the CUDA architecture. Experiments with two standard datasets were provided, the *cancer1* and the *mush1*. Our

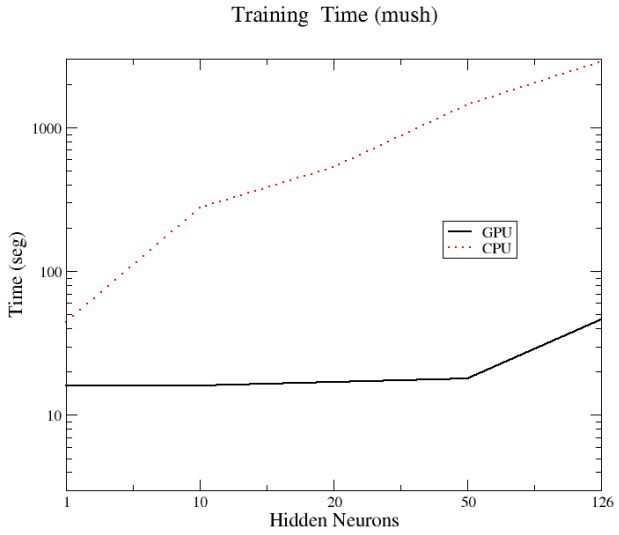

Figure 13. The number of hidden neurons vs the training time in seconds with the *mush1* dataset. A logarithmic time scale is used for illustration purposes.

tests indicate that the speedups of the order of 46x and 63x are possible for a neural network with one hidden layer. The performance gain is dependent on both, the feature vector size and the number of hidden neurons. Our future work considers the kernels optimization through the use of shared memory and modifications in the threads and blocks dimensions. Experiments using neural networks with more hidden layers is also considered as well as tests with other datasets.

ACKNOWLEDGMENT

Sierra-Canto wishes to thank PROMEP for supporting his research work.

REFERENCES

[1] F. Bernhard and R. Keriven, "Spiking Neurons on GPUs", International Conference on Computational Science, 2006.

[2] C. M. Bishop, "Pattern recognition and machine learning", Springer, New York, NY, USA, 2006.

[3] W. M. Caminhas, D. A. G. Vieira and J. A. Vasconcelos, "Parallel layer perceptron", Neurocomputing 55, pp. 771-778, 2003.

[4] A. Carpenter, "CUSVM: A CUDA Implementation of Support Vector Classification and Regression", patternsonscreen.net/cuSVMDesc.pdf, 2009.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer and K. Skadron, "A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA", Journal of parallel and distributed computing, 2008.

[6] A. Cristea and T. Okamoto, "A parallelization method for neural networks with weak connection design", International Symposium on High Performance Computing, 1997.

[7] L. Fausett, "Fundamentals of neural networks architectures, algorithm, and applications", Prentice Hall, New Jersey, USA,1997.

[8] T. Harada, "Real-time rigid body simulation on GPU (chapter 29) in GPU Gems 3", Addison Wesley, 2008.

[9] S. Haykin, "Neural networks a comprehensive approach", Prentice Hall, New Jersey, USA, 1999.

[10] H. Jang, A. Park and K. Jung, "Neural network implementation using CUDA and OpenMP", DICTA '08: Proceedings of the 2008 Digital Image Computing: Techniques and Applications, Washington, DC, USA, 2008.

[11] A. Lefohn, J. Kniss and J. Owens, "Implementing efficient parallel data structures on GPUs(chapter 32) in GPU Gems 2", Addison Wesley, 2006.

[12] D. L. Ly, V. Paprotski and D. Yen, "Neural Networks on GPUs: Restricted Boltzmann Machines", Technical Report, Department of Electrical and Computer Engineering, University of Toronto, 2008.

[13] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau and A. Veidenbaum, "Efficient simulation of large-scale spiking neural networks using CUDA graphics processors", IJCNN'09: Proceedings of the 2009 international joint conference on Neural Networks, Atlanta, Georgia, USA, 2009.

[14] S. Oh and K. Jung, "View-Point Insensitive Human Pose Recognition using Neural Network and CUDA", World Academy of Science, Engineering and Technology 60,2009.

[15] G. Poli and J. H. Saito, "Parallel Face Recognition Processing using Neocognitron Neural Network and GPU with CUDA High Performance Architecture", Face Recognition, Book edited by: Milos Oravec, ISBN: 978-953-307-060-5, INTECH, 2010.

[16] L. Prechelt, "PROBEN1 - a set of neural networks benchmark problems and benchmarking rules", 1994.

[17] D. E. Rumelhart and J. L. McClelland, "Parallel distributed processing: Explorations in the microstructure of cognition", MIT Press, Cambrige, 1986.

[18] U. Seiffert, "Artificial neural networks on massively parallel computer hardware", European Symposium on Artificial Neural Networks, Bruges, Belgium, April 2002.

[19] S. Suresh, S. N. Omkar and V. Mani, "Parallel implementation of back-propagation algorithm in networks of workstations", IEEE Transactions on Parallel and Distributed Systems, 16 (1), pp. 24-34, 2005.

[20] R. K. Thulasiram, R. M. Rahman and P. Thulasiraman, "Neural Network Training Algorithms on Parallel Architectures for Finance Applications", International Conference on Parallel Processing Workshops, Kaohsiung, Taiwan, October 2003.

[21] R. Uetz and S. Behnke, "Large-scale Object Recognition with CUDA-accelerated Hierarchical Neural Networks", In Proceedings of the 1st IEEE International Conference on Intelligent Computing and Intelligent Systems, 2009.