CONEM 2010
VI Congresso Nacional de Engenharia Mecânica

ABCM COMPROMETIDA COM A PROMOÇÃO DO DESENVOLVIMENTO DA ENGENHARIA E DAS CIÊNCIAS MECÂNICAS
ASSOCIAÇÃO BRASILEIRA DE ENGENHARIA E CIÊNCIAS MECÂNICAS

VI CONGRESSO NACIONAL DE ENGENHARIA MECÂNICA
VI NATIONAL CONGRESS OF MECHANICAL ENGINEERING
18 a 21 de agosto de 2010 – Campina Grande – Paraíba - Brasil
August 18 – 21, 2010 – Campina Grande – Paraíba – Brazil

PERFORMANCE COMPARISON OF TRAINING METHODS FOR MLP NEURAL NETWORKS IN GPU

Klaus Raizer, klausraizer@gmail.com[1]
Hugo Sakai Idagawa, idagawa@fem.unicamp.br[1]
Eurípedes Guilherme de Oliveira Nóbrega, egon@fem.unicamp.br[1]
Luiz Otávio Saraiva Ferreira, lotavio@fem.unicamp.br[1]

[1]Department of Computational Mechanics, Mechanical Engineering Faculty, State University of Campinas, Campinas, SP, Brazil

Abstract. *The present work describes the recent developments regarding the implementation of Feedforward Multi-Layer Perceptron (FFMLP) Neural Networks on GPU by using the nVIDIA CUDA programming language. At the Department of Mechanical Computation of the Mechanical Engineering Faculty a number of projects are being developed that make extensive use of classification techniques, such as system control and time series prediction. Since Neural Networks have proven to be highly efficient to solve a number of these problems, the development of an enhanced Neural Networks toolbox to assist at the development of software solutions was a natural step taken by the GPGPU (General Purpose GPU) group. This paper shortly describes how data is processed in a GPU and how recent work produced better results compared to those obtained until then. The structure of a feedforward multi-layer perceptron is explained and two training processes, batch and incremental, are described. Since a substantial part of both algorithms are composed of matricial operations, special care was taken at the development of a kernel suited for the current application. Performance gain for both versions is presented and compared using real applications as benchmark problems. Two benchmark problems were chosen: ECG anomaly identification and the traditional character recognition system. Results from each test are presented and commented. Based on the conclusions, future works on the project are proposed.*
Keywords: *Neural Networks, GPU, CUDA, backpropagation, ECG*

1. INTRODUCTION

Motivated by an ever increasing demand for high definition and real-time 3D graphics, GPUs (Graphics Processing Units) have become a platform with huge processing capacity due to their inherent parallelism and high bandwidth memory. All this processing power allowed the development of a new area in the High Performance Computing (HPC) field, known as General-purpose Computing on GPUs (GPGPU), which takes advantage of the highly-parallel architecture of GPUs to solve numeric intensive problems. Some examples of these problems are found in Cryptography, FFT, 2D and 3D segmentation, image processing, computer vision, many problems in the bioinformatics field and Artificial Intelligence techniques like Genetic Algorithms and Artificial Neural Networks (ANN) (Owens et al., 2007).

This work relates the implementation in GPU of a specific, but with broad applications, type of Artificial Neural Network called Feedforward Multilayer Perceptron (FFMLP).

1.1 CUDA and Graphics Processing Unit

A high demand for faster processing of 3D and high definition graphics induced the production of the current programmable GPUs, which have huge processing power and high parallelism. As a matter of fact, the increase in Floating-Point Operations per Second (Flops) for the GPU is faster than for CPU. Recent numbers show that GPU has a performance of more than seven times GFlops/second than a current CPU (NVIDIA, 2008). The reason for this discrepancy is that GPUs are specialized in highly parallel and computationally intensive processes. This means that the GPU is projected in a way that more transistors are used to process data instead of being used to store data or for flux control. The structural difference between a CPU and a GPU can be seen on Fig. 1.

NVIDIA exposes its GPU architecture as a group of multiprocessors, each one comprised of a group of 8 CUDA cores (each one capable of executing arithmetic instructions), a control unit and a small cache. This architecture allows the GPU to operate similarly to a vector processor capable of executing instructions in a SIMD (Single instruction, multiple data) fashion (Volkov and Demmel, 2008).

Until very recently the use of GPUs for numerical calculations demanded the usage of specific application program-

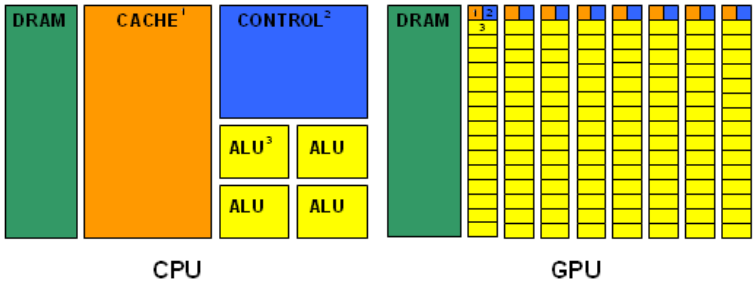

Figure 1. Structural differences between a CPU and a GPU

ming interfaces (APIs) used in Computer Graphics programming, like OpenGL and DirectX. These APIs were inadequate for processing general applications, which demanded the programmers to adapt their algorithms to them. This complicated the access to this technology because these programmers weren't necessarily experts in graphics programming: while 3D programmers talk in terms of shaders, textures and fragments; specialists in parallel programming talk about streams, kernels and threads.

In 2006, NVIDIA changed this paradigm by introducing CUDA (Compute Unified Device Architecture) which is a parallel computing architecture with a new programming model that takes advantage of the GPU programmable parallel hardware (NVIDIA, 2008). CUDA also extends the C programming language, allowing programmers to use functions named kernel to call a process a number of times concurrently, being each data processed by an individual thread. A diagram of how the inner structure of a parallel CUDA program is hierarchically organized can be seen on Fig. 2.

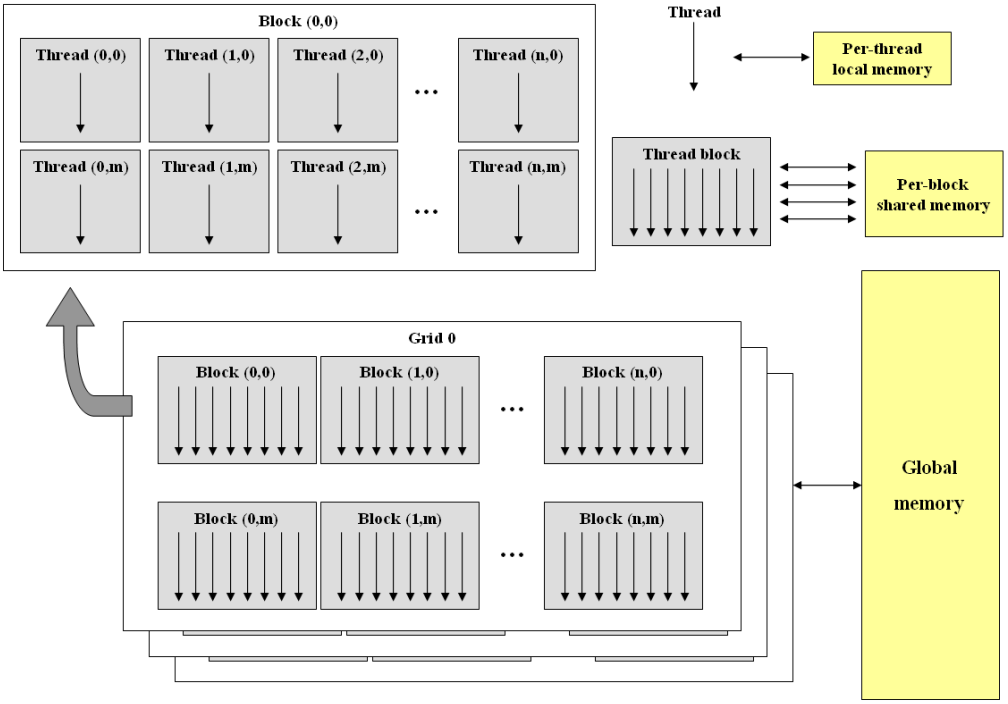

Figure 2. Organization and hierarchy of threads, blocks and grids

1.2 Artificial Neural Networks

Neural networks are massively parallel distributed processors, composed by simple processing units called neurons, that can store experiential knowledge through a learning process (Haykin, 1998).

Beyond being able to learn from examples, an important feature from a neural network is its ability to generalize, by producing reasonable outputs for inputs distinct from those used in the learning process.

As previously mentioned, a neural network is composed by a number of sub-components called neurons. Some of the major components of a biological neuron are dendrites, cell body, axon and synapses, the places where connections

between neurons occur.

Nerve impulses through these synapses can result in local changes in the potential in the cell body of the receiving neuron. These input potentials can spread through the main body of the cell, being summed at the axon hillock. If the amount of depolarization at the axon hillock is sufficient, an action potential is generated and travels then through the axon (Freeman and Skapura, 1991).

1.3 Forward phase

It is possible to determine a mathematical model that behaves like the original biological neuron, such as the one in Fig. 3. These artificial neurons can then be organized into layers producing a structure that behaves similarly to a biological neural network.

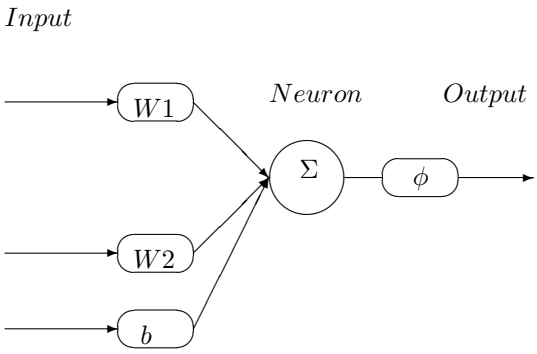

Figure 3. Artificial model of a Neuron.

As a rough approximation of the workings of a biological neuron, the artificial neuron consists of a weighted sum of its inputs, followed by the application of an activation function, as can be seen in Fig. 3.

The activation function ϕ defines the output of a neuron and it can be of many different kinds. Some of the most used activation functions can be seen in Fig. 4. In this figure, the dash-dotted blue line corresponds to a sigmoid function, also known as the logistic function, the dashed red line corresponds to a hyperbolic tangent function and the solid black line corresponds to a linear activation function.

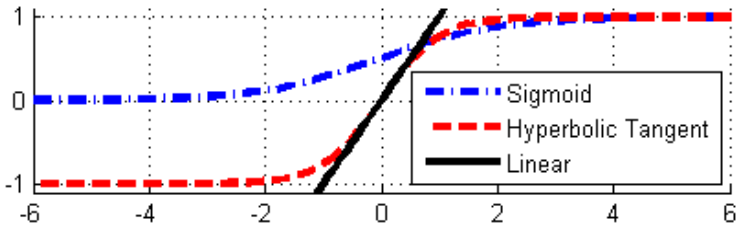

Figure 4. Activation Functions: Sigmoid, Hyperbolic Tangent and Linear.

The logistic function, with its output ranging from 0 to +1, is the most common activation function and it was the one used in the present work.

These artificial neurons can then be organized in layers. When there is only one layer, the resulting neural network is called a *single-layer network*. One layer's output can serve as another's input, in which case a *multi-layer network*, like the one in Fig. 5 is formed.

It is mathematically more compact to express the structure seen in Fig. 5 as a series of matrix and vectors' operations. The output of one layer can then be written as seen in Eq. 1.

$$s = \phi(W \cdot i + b) \tag{1}$$

Being ϕ the activation function, i the input vector for the current layer, b the bias vector and W the weight matrix, as described in Eq. 2.

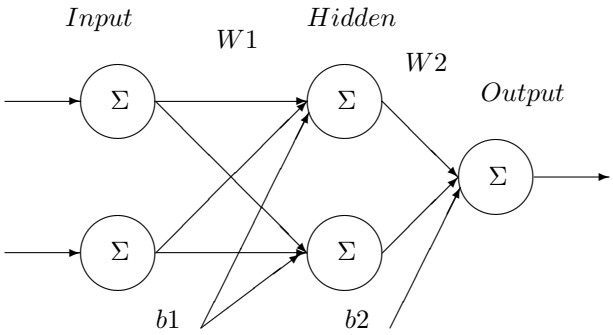

Figure 5. Neurons organized in layers, with one layer's output serving as the next one's input

$$W = \begin{bmatrix} W_{1,1} & W_{1,2} & \cdots & W_{1,nInputs} \\ W_{2,1} & W_{2,2} & \cdots & W_{2,nInputs} \\ \vdots & \vdots & \ddots & \vdots \\ W_{nNeurons,1} & W_{nNeurons,2} & \cdots & W_{nNeurons,nInputs} \end{bmatrix} \tag{2}$$

Bias is an input fixed as 1 and has its own weight. As can be seen on the weight matrix, lines on W represent weights for a particular neuron, being the number of columns equivalent to the number of neurons and the number of lines equivalent to the number of inputs.

1.4 Training

In order for an artificial neural network to perform the task it was meant to perform, its structure and weights must be defined. There are a number of ways to build the desired input-output mapping for a neural network, ranging from genetic algorithms to the traditional gradient descent method. In this work the backpropagation algorithm was implemented, based on the above mentioned gradient descent method to find a minimum in the error surface such as the one seen in Fig. 6.

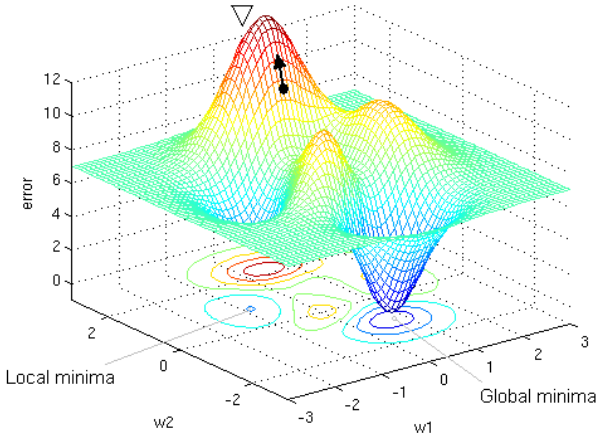

Figure 6. Error surface in the weigth space for two weights: the arrow shows gradient's ∇ direction

Figure 6 shows the specific case of a single neuron with only two weights. For each possible value they might assume, a different result comes out from the output for a given input. This result is initially different from the expected output, resulting in an *error*, which can be measured as the difference between the expected output and the produced output.

The overall objective is to minimize the total error of the output vector so the produced output gets closer to the expected. The usual function to be minimized can be seen in Eq. 3 (Haykin, 1998), with N being the total number of elements in the output vector.

$$E(n) = \frac{1}{2} \cdot \sum_{k=1}^{N} error_k(n)^2 \tag{3}$$

Weights forming the network must be corrected in order to make the error seen in Eq. 3 as small as possible. One way of doing so is by taking the gradient of the error surface, like the one seen in Fig. 6 and correcting all the weights in the opposite direction, which constitutes a *gradient descent* optimization technique.

The total weight correction can be found by using Eq. 4 (Haykin, 1998).

$$\Delta W_{kj}(n) = \eta \cdot error_k(n) \cdot \frac{df(wSum)}{dwSum} \cdot I_j(n) \tag{4}$$

Being η a positive constant called *learning rate*, *wSum* the weighted sum at the given neuron, **k** the current neuron, **j** the current input and the term $\frac{df(wSum)}{dwSum}$ the derivative of the activation function.

Since there is no direct access to the output error at a given hidden neuron, we must estimate it by applying Eq. 5. In essence, this backprograpation of errors is similar to the forward propagation seen in Section 1.3 The major difference being that the error is backpropagated through output connections from a unit, rather than through input connections (Freeman and Skapura, 1991).

$$error_k(n) = \sum_{l=1}^{L} error_l(n) \cdot W_{kl}(n) \tag{5}$$

With **k** as the neuron of the previous layer, l the neuron of the current layer and L the total number of neurons in the current layer. In other words, the estimated error of a given neuron at the first layer will be defined by the weighted sum of the errors existent in all the neurons of the next layer that this particular neuron is connected with.

Each input produces a correction in the neural network's weights. A number of inputs, called training set, comprising the problem to be solved are presented to the neural network during the training phase. Going through all inputs in the training set completes what is called an epoch.

2. METHODS

When implementing the training method described in Section 1.4 there are two basic ways to update the weights: batch mode and incremental mode.

The difference between these two training modes is the fact that in incremental mode, for each input vector presented to the network, there is an immediate correction of this network's weights. As shown in Fig. 7 this process occurs in two loops, the first one loops through epochs of training while the second one loops through all samples of inputs comprising the training set.

The feedforward step is followed by the backpropagation of errors, which allows correcting the weights towards a smaller output error. The dotted box around those three processes identify them as being parallelizable, and each one was also implemented in GPU.

In batch mode on the other hand, all inputs comprising the problem training set are presented to the network, each input produces weights' correction and all these corrections are accumulated, being applied only at the end of the training epoch.

Figure 8 describes how the batch mode was implemented. Since there is no need to immediately perform weight's correction for each presented input. The second loop seen in Fig. 7 ceases to exist, being performed at once by a matrix multiplication kernel.

2.1 Benchmark

In order to evaluate the differences between training methods two benchmark problems were chosen: ECG pulse classification with neural networks and handwritten character recognition. In this section both problems will be briefly explained.

2.1.1 ECG classification

The ECG (electrocardiogram) is a record of the human heart's electrical activity. ECG signals are measured by a set of electrodes attached to the human body, in order to detect variations in the electrical potential of the heart. In the *12-lead* technique, 12 different derivations are produced, each representing a particular view of the heart in action.

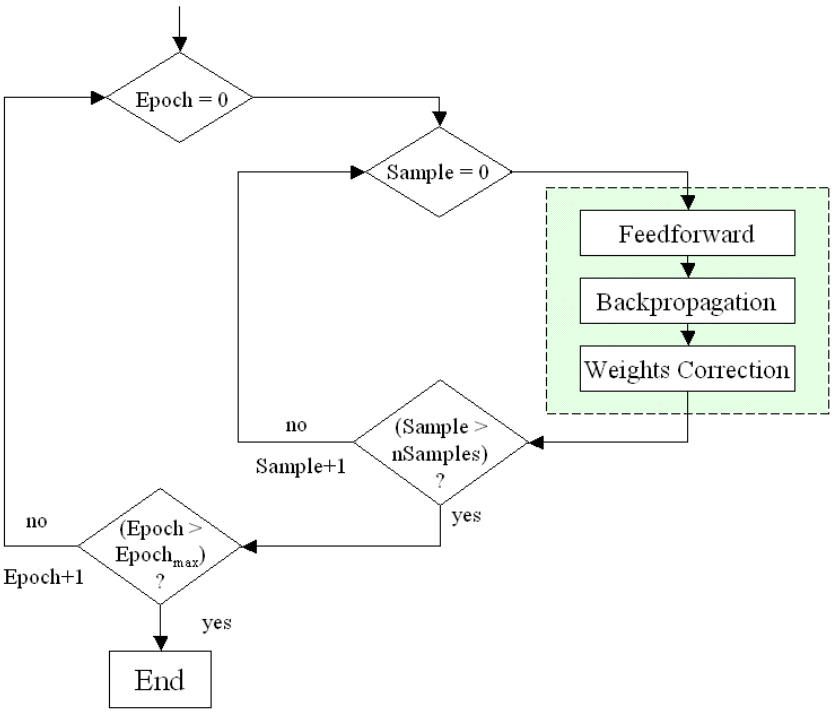

Figure 7. Incremental training diagram

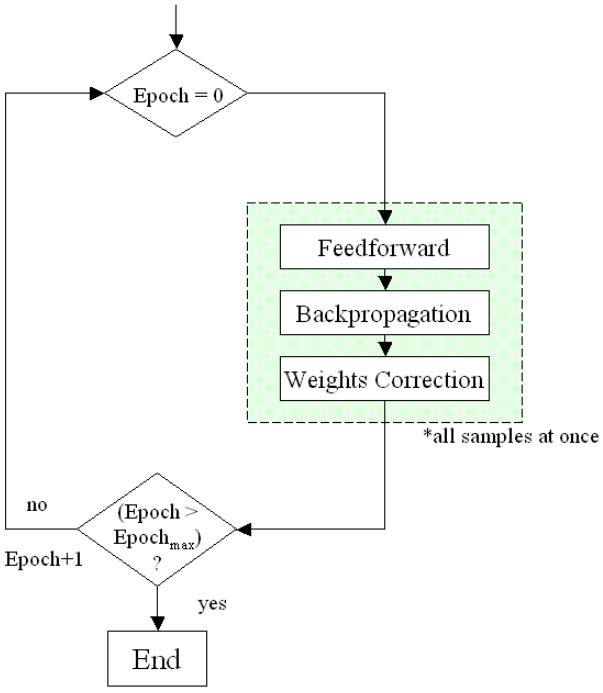

Figure 8. Batch training diagram

Since each process within the heart, produces its own ECG signature, it is possible to detect a number of heart diseases by identifying deviations in the ECG signal.

At the Department of Mechanical Computation there is an ongoing project aimed at developing an ECG signal analysis system, whose classification layer is composed by a MLP artificial neural network (Raizer et al., 2010). Each ECG pulse, corresponding to a single heart beat, is isolated and a 36 elements normalized vector of features is extracted from it using the discrete wavelet transform (DWT) (Mallat et al., 1989).

Signals were obtained at the PhysioBank (physiologic signal archives for biomedical research) database and were taken from the MIT-BIH Arrhythmia set of signals (Goldberger et al., e 13).

The objective is to classify each heart pulse as being either normal, premature ventricular contraction, right bundle branch Block or left bundle branch block (Saxena et al., 2002).

The ECG classification problem renders a network with 37 inputs, comprised of 36 features plus a bias (automatically inserted into the input vector) and 4 neurons at the output, so each neuron would be responsible for classifying a particular cardiopathy. The size of the hidden layer was set as having 8, 16, 32, 64, 128, 256, 512 and 1024 neurons to test the training performance in both CPU and GPU.

2.1.2 Handwritten character recognition

In order to investigate the performances of the training algorithms for problems with a higher number of inputs, a handwritten character recognition system was developed for this work. Data was obtained at the UCI Machine Learning Repository (Asuncion and Newman, 2007), from the Semeion Handwritten Digit Data Set. The data set is comprised of 1593 handwritten digits, ranging from 0 to 9, scanned from 80 different persons. Each digit was placed in a 16x16 grid, equivalent to a 256 elements vector in gray scale. Each pixel was then changed to a Boolean value (1/0), producing what can be seen in Fig. 9.

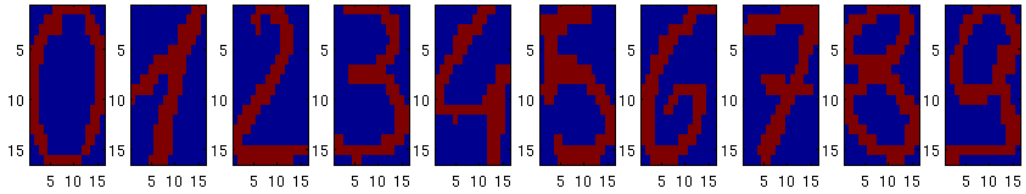

Figure 9. Handwritten numbers, ranging from 0 to 9

The neural network is set as having 257 inputs (256 plus 1 bias) and 10 outputs. Each output represents a specific number and at a given input only one out of ten outputs should have a value near *one* while the other 9 outputs' values should be near *zero*.

3. RESULTS

All versions of the algorithms were implemented in C for the CPU hardware and in CUDA for the GPU. Hardware configurations are:

CPU	GPU
OS: Linux Ubuntu version 9.04: kernel 2.628-15	GeForce 8800GT
Memory: 4GB RAM	Memory: 1024Mb
Processor: Intel Core 2 Dual E8400@3.00GHz	GPU clock: 600MHz
	Memory clock: 900MHz

For small problems, those with a reduced number of neurons, there is no advantage of the GPU version over the CPU one for incremental mode, as can be observed in Fig. 10(a). That happens because small problems leave part of the processing units without use. In batch training however, since all training input vectors are processed at the same time, the GPU version of the algorithm shows a speed-up gain right from the start, as can be seen in Fig. 10(b).

A downward slope was observed in the speed-up for the incremental case, as seen in Fig. 10(b). As the number of neurons in the second layer increases beyond what is required for the mapping being performed, it becomes easier for the training method to find a valley in the error function which satisfies the error requirements. Because of this the number of epochs required to achieve those error requirements is reduced, interfering with the calculation of the time per epoch ratio.

For the character recognition problem, the behaviour observed in Fig. 11(a) presented the same pattern seen in Fig. 10(a). The batch version implemented in GPU presented an increasing gain in performance, being faster than its incremental counterpart. In the incremental case, the GPU version is advantageous only for a hidden layer with more than 240 neurons, as can also be seen in Fig. 11(b). The batch version analysis has produced a gain around 2.5 times for larger networks but has also shown an even greater speed-up for smaller networks. This result was different from expected, and could be a result of how blocks are arranged during kernel execution. The nature of this particular phenomenon will be further investigated in the next phase of the project.

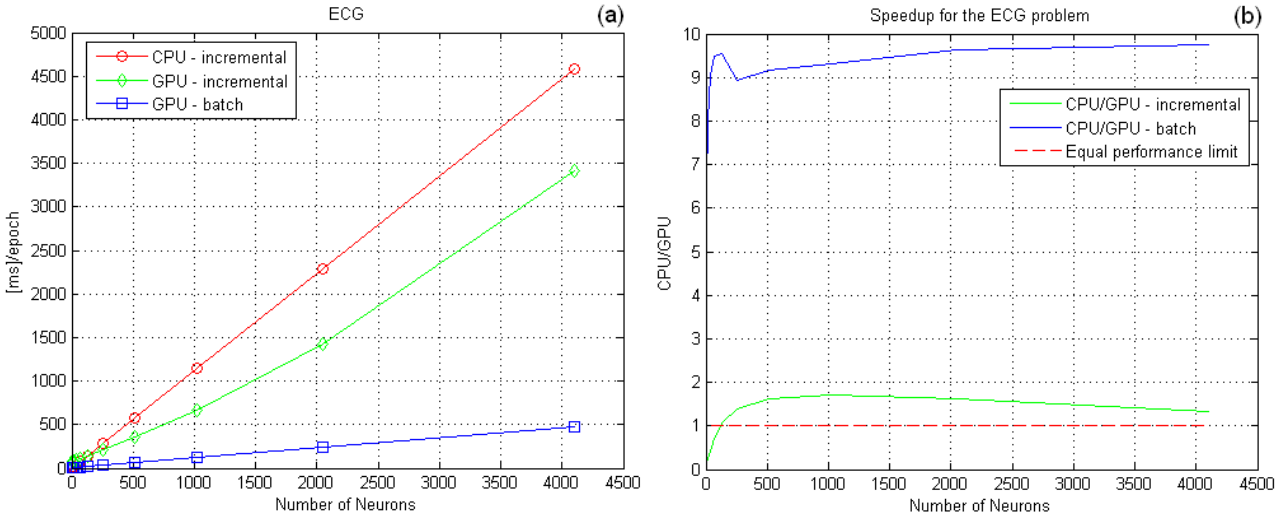

Figure 10. Performance comparison between CPU and GPU (a) and CPU speed over GPU speed (b) for the ECG problem

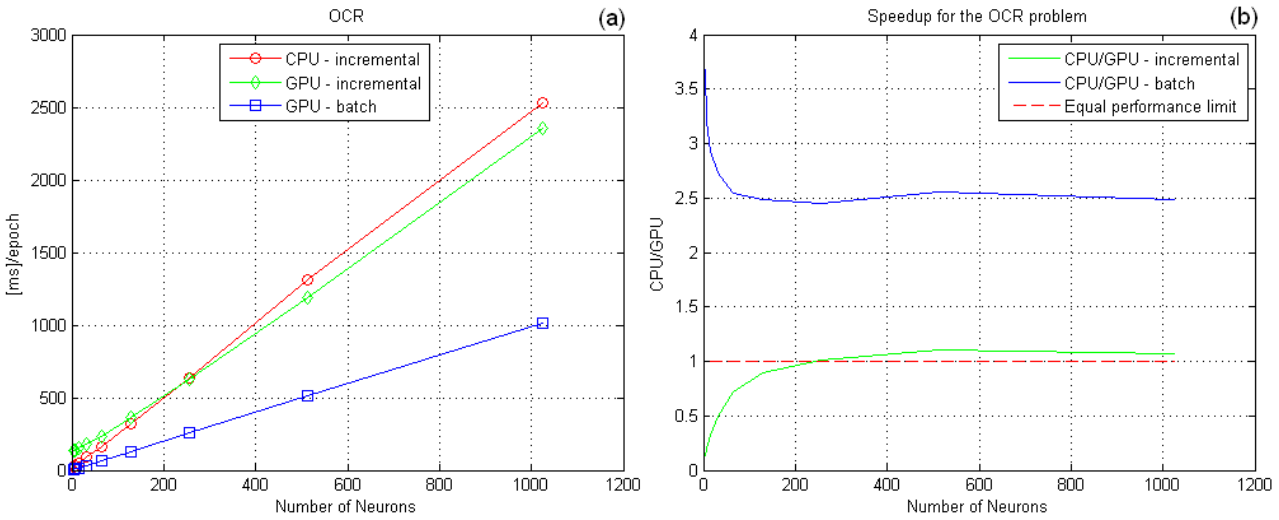

Figure 11. Performance comparison between CPU and GPU (a) and CPU speed over GPU speed (b) for the OCR problem

4. CONCLUSIONS

As expected, batch training mode takes more advantage from the GPU hardware than the incremental mode, since a greater number of matrix operations are performed per epoch. This resulted from eliminating the second CPU loop, turning it into a single matrix operation.

Results for the ECG problem showed a speed-up of almost 10 times, an expressive improvement compared to those obtained in its previous implementation, which produced a speed-up of 3 to 4 times. This resulted from improving the matrix multiplication kernel used in (Raizer et al., 2009), as commented in Section 1.1

Future work will focus on developing a more robust training algorithm for the Artificial Neural Networks toolbox, and on turning its use more user-friendly.

5. ACKNOWLEDGEMENTS

We would like to thank CAPES for the financial support during the development of this project, and also the Department of Mechanical Computation at the Mechanical Engineering Faculty for the opportunity and the resources to develop this work.

6. REFERENCES

Asuncion, A. and Newman, D. (2007). UCI machine learning repository.

Freeman, J. A. and Skapura, D. M. (1991). *Neural networks: algorithms, applications, and programming techniques*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

Goldberger, A. L., Amaral, L. A. N., Glass, L., Hausdorff, J. M., Ivanov, P. C., Mark, R. G., Mietus, J. E., Moody, G. B., Peng, C.-K., and Stanley, H. E. (2000 (June 13)). PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220. Circulation Electronic Pages: http://circ.ahajournals.org/cgi/content/full/101/23/e215.

Haykin, S. (1998). Neural networks: a comprehensive foundation. *New Jersey: Prentice Hall*.

Mallat, S. et al. (1989). A theory for multiresolution signal decomposition: The wavelet representation. *IEEE transactions on pattern analysis and machine intelligence*, 11(7):674–693.

NVIDIA (2008). *NVIDIA CUDA Programming Guide 2.0*. nVIDIA.

Owens, J., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, E., and Purcell, T. (2007). A survey of general-purpose computation on graphics hardware computer graphics. *F*, 26(1):80–113.

Raizer, K., Idagawa, H., de Oliveira Nóbrega, E., and Ferreira, L. (2009). Training and Applying a Feedforward Multilayer Neural Network in GPU. *Proceedings of the 30th Iberian-Latin-American Congress on Computational Methods in Engineering*.

Raizer, K., Nóbrega, E. G. O., Ferreira, L. O. S., and Costa, E. T. (2010). Análise de sinais de ECG com o uso de Wavelets e Redes Neurais em FPGA. Master's thesis, Universidade Estadual de Campinas (UNICAMP) - Faculty of Mechanical Engineering, Brazil - SP.

Saxena, S., Kumar, V., and Hamde, S. (2002). Feature extraction from ECG signals using wavelet transforms for disease diagnostics. *International Journal of Systems Science*, 33(13):1073–1085.

Volkov, V. and Demmel, J. (2008). Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Press.

7. RESPONSIBILITY NOTICE

The authors are the only responsible for the printed material included in this paper.