Speeding Up the Training of Neural Networks with CUDA Technology

Daniel Salles Chevitarese, Dilza Szwarcman, and Marley Vellasco*

Department of Electrical Engineering, Pontifical Catholic University,
Rua Marquês de São Vicente, 225, Gávea - Rio de Janeiro, Brazil
daniel@ele.puc-rio.br
http://www.puc-rio.br

Abstract. Training feed-forward neural networks can take a long time when there is a large amount of data to be used, even when training with more efficient algorithms like Levenberg-Marquardt. Parallel architectures have been a common solution in the area of high performance computing, since the technology used in current processors is reaching the limits of speed. An architecture that has been gaining popularity is the GPGPU (General-Purpose computing on Graphics Processing Units), which has received large investments from companies such as NVIDIA that introduced CUDA (Compute Unified Device Architecture) technology. This paper proposes a faster implementation of neural networks training with Levenberg-Marquardt algorithm using CUDA. The results obtained demonstrate that the whole training time can be almost 30 times shorter than code using Intel Math Library (MKL). A case study for classifying electrical company customers is presented.

Keywords: Artificial Neural Networks, Software Engineering, High Performance Com-puting, GPGPU, CUDA.

1 Introduction

Neural networks are very useful for solving complex problems (pattern recognition, forecasting, classification) and there are already many software libraries that support the modeling, creation, training and testing of various types of known networks. However, the available libraries present limitations when the problem size or complexity exceeds certain threshold, such as the case when the database used in the early stages of training, validation and testing contains a huge amount of information (patterns/attributes).

Besides the database size, there is also a tradeoff between the complexity of the algorithm used for training and the number of iterations needed to reach the network's performance goal, which greatly affects the total training time. For example, if a neural network is trained by an algorithm of back propagation with gradient descent, the cost of each step is relatively small; however, many steps are required to train the network. On the other hand, a greater order gradient algorithm requires much less iterations, but with a much greater computational cost for each iteration [5].

There are already several studies using graphics cards to propagate input data through feedforward neural networks, but the learning algorithm is not implemented in CUDA.

* IEEE Senior Member.

L. Rutkowski et al. (Eds.): ICAISC 2012, Part I, LNCS 7267, pp. 30–38, 2012.

For example, [2] and [7] have used neural networks for character recognition. In the second paper, the program implemented in CUDA was almost six times faster than the same program running on the CPU. In [1] a program for training neural networks was implemented using the gradient descent algorithm, which was compared to Matlab and resulted in tens of times faster.

This work proposes a faster implementation of neural networks training with Levenberg - Marquardt algorithm (LMA) using CUDA and compares its performance with an implementation using Intel MKL [6], a math library which is known for its outstanding efficiency.

The results obtained with the execution on the GPU of the training phase using LMA were very promising. Although only part of the algorithm was processed on the graphics card, again of almost 30 times faster was obtained when compared with the same algorithm running with the help of MKL.

This paper has been organized as follows: Section 2 presents an overview of CUDA Architecture; Section 3 describes the architecture proposed for solving the problem; Section 4 demonstrates the experiments performed and lastly, Section 5 presents the conclusions drawn from this work.

2 GPGPU and CUDA

While parallel solutions are becoming more common, graphics cards are becoming powerful computers and highly parallel, mostly because of the digital entertainment industry and its demand for high definition graphics. The reason for the huge discrepancy between conventional processors and graphics processors is that a GPU (Graphics Processing Units), unlike the CPU, uses more transistors for processing than for flow control or cache memory [8].

The GPU architecture is better suited to problems whose data can be broken into smaller pieces and be processed in parallel. As these pieces are processed by the same program and at the same time, they do not require a sophisticated flow control. Moreover, as there are many arithmetic calculations to be performed, the latency of memory access is diminished by data buffering instead of the use of great cache memories.

The NVIDIA architecture has three types of abstraction: the hierarchy of thread groups, the shared memories and synchronization barriers. This architecture makes the learning time relatively small and requires few extensions to programming languages. Moreover, the abstractions provide parallelism both in data and in threads, requiring, from the programmer, only a simple division of tasks and data [8].

Another advantage of the abstractions is that they help the developer divide the problem into smaller tasks that can be solved independently and, therefore, in parallel. This decomposition is made so as the threads can cooperate to solve the subtasks, and, at the same time, make scalability possible, since the threads can be scheduled to be resolved in any available core. A program compiled in CUDA, which is also called kernel, may, in that case, run on a machine regardless of the number of processors, which will be checked at runtime [8]. Thus, the programming model comprises several types of clients, supporting cards with different number of processors (this number can vary from ten to thousands of processors).

3 The Proposed LMA in CUDA

In this work, neural network training is carried out by the Levenberg - Marquardt Algorithm, which calculates the neuron's errors in a way equivalent to conventional back - propagation, but based on the Jacobian [4]. In order to combine a reduced number of epochs with low time cost iteration, this study proposes a new design of a Levenberg - Marquardt training which runs on a GPU (Graphic Processor Unit) supporting CUDA. The new model uses the computational power of graphics cards to calculate the critical point of the training algorithm, that is, the change in the network's weights given in Equation (1) [4]. Its processing time generally represents more than 70% of the total training time if the training data set is small, and can reach almost 90% of the time for larger sets. This is because Equation (1) includes the computation of a Jacobian matrix of size (w, p), where w is the number of network weights and p is the number of patterns. In this equation, x is the weights vector of the network and $\underline{e}\,(\underline{x})$ is the error vector, while μ is a parameter that controls the balance between speed and stability. This matrix can easily contain millions of elements even if the training data includes just a few thousand patterns.

$$\triangle x = -\left[J\,(\underline{x})\,J^T\,(\underline{x}) + \mu I\right]^{-1} J^T\,(\underline{x})\,\underline{e}\,(\underline{x}) \tag{1}$$

The use of graphics cards has been proved promising, since the number of processing cores on a single card can be up to hundreds. In addition, the time to manage threads in conventional languages can cost more than 10% of the total training time, while the NVIDIA architecture can handle CUDA threads without additional cost. Another positive point of NVIDIA technology is the possibility of working with multiple graphics cards, allowing the execution of several concurrent trainings.

Initially, this work proposed the use of the graphics card to calculate the Jacobian square matrix $(J(\underline{x})J^T(\underline{x}))$, which is a part of Equation (1). For that matter, a function (from NVIDIA) copies all data to the global memory. Once in global memory, a kernel calculates it and, after that, another CUDA function transfers the result back to the RAM of the CPU. The second model was to create a kernel that calculates the whole equation in the GPU, so the vector with the weights variation is obtained directly from it, as shown in Figure 1. In this figure, the black box indicates the training starting point or a new epoch, including the patterns propagation through the neural network. The following stages calculate the squared errors of the network, as shown in equation (2).

$$E\,(x) = \sum_{i=1}^{N} e_i^2\,(\underline{x}) \tag{2}$$

In order to calculate $\triangle x$ from the Jacobian matrix, Equation (1) has been divided in three major steps: the calculation of $(J\,(\underline{x})\,J^T\,(\underline{x}) + \mu I)$, the calculation of the inverse resultant matrix by Gaussian Decomposition, and the multiplication of the previous result with $J^T\,(\underline{x})\,\underline{e}\,(\underline{x})$. To calculate the first and the third steps, this work used the library CUBLAS from NVIDIA, which implements levels 1, 2 and 3 of the known library BLAS (Basic Linear Algebra Subroutines) to run in GPU. The second step of this kernel is described below:

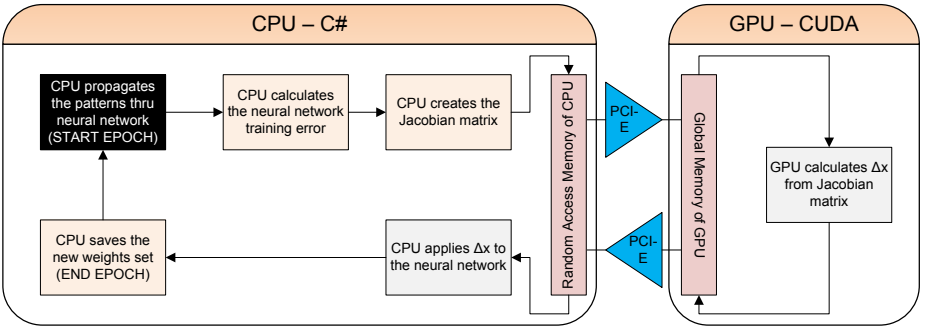

Fig. 1. Proposed model, which performs a major portion of the code in the graphics card. In this picture, PCI-E means PCI-Express bus.

1. Divide the matrix $(J\,(\underline{x})\,J^T\,(\underline{x}))$ in sub matrixes (squared) in order to have these parts in blocks of shared memory. Using this technique, the global memory is read and written only once per sub matrix;
2. The Gaussian Decomposition is applied on those sub matrices and the pivots are calculated. The pivots here have the same function as on LU decomposition, for example;
3. Using the pivots, the adjacent rows are updated. In this step it is necessary to synchronize all blocks, because the pivots calculated in the first block are also used to update the other block and generates its respective pivots;
4. At the end, all elements above the principal diagonal are equal to zero.

4 Main Results

This section presents 7 experiments where the proposed model was used to train networks and ensembles of neural networks. On these experiments, ensembles of neural networks were used to classify customers of a Brazilian electricity distribution company (Light) as regular or irregular. Four types of NVIDIA graphics cards were used, as shown in Table 1, where Gflop means how many floating-point operations, in billions, can be performed per second, Number of SP means the number of Stream Processors onboard, and CUDA capability means which version of the architecture is supported by the graphics card (Min 1.1 and Max 1.3) [8]. Computer specification I is an Athlon 6000 with 1GB RAM and a GeForce 8400 GS; specification II is an Athlon 6000 with 2GB RAM and a GeForce 8800 GT; specification III is an Athlon 6000 with 2GB RAM; and a GeForce 260 GTX and specification IV is a Phenom II with 16GB RAM and four Tesla c1060.

4.1 Problem Description

Light is a company with 3.79 million consumers divided in 5 regions (East, West, Coastal, Interior, Lower Region) in 31 cities of Rio de Janeiro. The company loses

Table 1. Computer configurations used in the experiments

Computer Spec	Number of SP	Memory size (MB)	Peak (Gflop/s)	CUDA Capability
I	8	256	2, 5	1.1
II	112	512	194	1.1
III	192	896	310	1.3
IV	4x240	4x4096	4x340	1.3

more than US$400 million with annual non-technical losses [3]. In order to put pressure on this kind of companies to adjust their prices through the reduction of losses, the Regulatory Agency for Electric Energy of Brazil (ANEEL) introduced new rules to limit the amount of non-technical losses, fraud and theft that can be charged to the customer. In addition, ANEEL is treating the issue as a priority and intends to adopt a policy of no tolerance [3], forcing companies to clearly specify goals for reducing these losses.

Currently, Light uses a set of methodologies and is associated with a reporting service to help identify low-voltage customers suspected of committing any type of fraud. These customers are classified as suspect by these methodologies and a specialist company selects a particular set of customers to be inspected. Through this process, Light has obtained an average of 25% successes in the verification of fraudulent customers [3]. In this experiment, a prototype of an intelligent computer system for identifying the fraudulent customer profile was developed, providing information to help select customers to inspect, increasing the effectiveness of energy recovery actions. In this experiment, we developed a prototype of an intelligent computer system for identifying the customer profile fraudsters, providing information to help a selection of customers to inspect, increasing the productivity of recovery actions energy.

4.2 System Architecture

The system structure is divided in three modules: (a) Preprocessing, (b) Filtering and (c) Classification. The module (a) includes data cleaning, where duplicated or corrupted data, missing values and outliers are removed, codification and normalization of categorical attributes and selection of those attributes. The Filtering (b) and Classification (c) modules use Ensembles composed of five MLP neural networks and 28 input attributes, one hidden layer and one neuron in the output layer. The whole process is shown in Figure 2.

This case study was chosen to prove the efficiency of the proposed model because of the high computational cost it demands. The Filtering module has to process all database in order to provide a new base with less noise to be processed by the Classification module, which can take almost 3 days (tested environment).

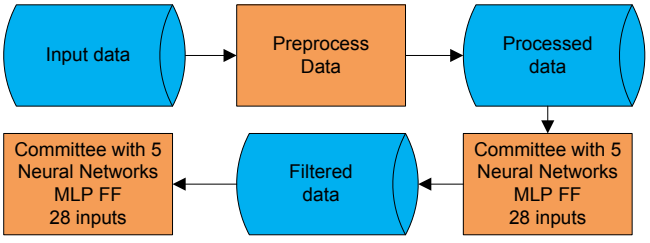

Fig. 2. Overview of system architecture

4.3 Results

The results of this experiment refer only to the performance of the two architectures described earlier, one where a $\triangle x$ calculation running on the CPU making the use of MKL library, and another where the most expensive code is handled by the video card. To calculate, approximately, the training time of one network, you have to get the showing times and multiply by the number of epochs, by ± 50 (adjust of μ) and by 10% (the time to process the rest of the code). The times in the tables refer to the training time of a ensemble and are in milliseconds.

Since commercial customers of all regions have similar profiles, only one ensemble was created to classify all of them, independently of their region. Other customers, such as residencial and industrial, where divided by region, with a different ensemble for each of the five existent regions. Table 2 presents the configuration for each region as well as for commercial customers.

Table 2. Regions properties

Region	Number of customers	Number of hidden neurons	Number of epochs	Jacobian Size (bytes)
Commercial	53, 307	12	50	148, 406, 688
East	98, 983	14	50	321, 496, 784
West	123, 272	15	50	428, 986, 560
Coastal	52, 217	10	50	212, 143, 440
Interior	27, 353	12	50	76, 150, 752
Lower	203, 704	14	50	661, 630, 592

The results in Table 3 show the training times using the Intel MKL and the times using the CUDA architecture. The number of neurons in hidden layer was defined by testing many other possibilities and the number of epochs was defined as 50, but the algorithm always get the network with the lowest validation error. This technique has a similar effect of early stopping. In addition, the speedup is shown in the last column meaning how many times one architecture is faster than the other. Some times could not be measured, because the matrixes' size was bigger than the graphic board memory.

Table 3. Results

Region	Configuration	MKL Time	CUDA Time	CUDA Speedup
Commercial	I	$4,855$	–	–
	II	$3,535$	843	4.18
	III	$3,535$	670	5.26
	IV	$1,906$	213	8.95
East	I	$9,744$	–	–
	II	$7,878$	–	–
	III	$7,878$	889	8.27
	IV	$4,427$	232	19.08
West	I	$13,275$	–	–
	II	$11,372$	$1,140$	9.98
	III	$11,372$	$1,138$	9.99
	IV	$6,203$	274	22.64
Coastal	I	$3,217$	$1,922$	1.67
	II	$2,542$	499	3.53
	III	$2,542$	312	5.45
	IV	$1,475$	202	4.91
Interior	I	$3,217$	$1,922$	1.67
	II	$1,700$	499	3.53
	III	$1,762$	312	5.45
	IV	992	202	4.91
Lower	I	$16,200$	–	–
	II	$16,255$	–	–
	III	$16,255$	$1,482$	10.97
	IV	$9,363$	333	28.12

The results of training the network with the commercial customer database already show a considerable difference between CPU and GPU, even though the base is not very large. Configuration I could not be assessed because the graphics card has failed reading one of the matrices. This error may be caused by the operating system that restarts the graphics card driver after 3 seconds if GPU stays unresponsive during that time. In Microsoft Windows, this feature is within the WDDM (Microsoft Display Driver Model) and is known as TDR (Timeout Detection and Recovery). The same problem occurred with East, West and Lower, but it was expected since those database are bigger than the commercial customer base. An important point that can be noticed in the results for East customers is the increased difference between the training time on the CPU and on the GPU. One of the factors that can explain this result is that, by doing the calculation for $\triangle x$ in the GPU, the CPU needs to transfer all matrices to global memory on graphic card every epoch. It is also important to notice the difference between the CPU and GPU clocks; some CPU has a frequency almost five times greater than some GPU. In the experiments conducted in this paper, this difference reached three times.

The customer database of region West is even larger than the customer base of region East. As was expected, the difference of training times between CPU and GPU has increased. However, the first GPU configuration could not be used due to memory size. As indicated above, this graphic card has only 256 MB of global memory and

the required memory to calculate $\triangle x$ is at least 450 MB. The customer base of region Coastal presents an important result related to the speedup of the first configuration. The training time in CPU and on the GPU are virtually tied. The causes have been described previously, but this fact indicates that an assessment regarding the problem complexity must be made to verify if the architecture running on a GPU is recommended.

When running the training base for customers of region Interior, even configuration I shows a performance gain of almost 200% higher compared to the CPU. Moreover, the cost of the graphics card used in configuration I is approximately US$ 30, while the computer can cost more than US$ 300.

The largest customer base is the one of Lower region, with more than 200,000 customers. On this last experiment the great difference between configurations III and IV can be noticed, as configurations I and II could not be assessed due to memory size. The training time of configuration IV is almost three times faster than the time of configuration III, and the reason for this difference is the number of processors inside these two GPUs, besides the number of records processed.

5 Conclusions

In this paper, a training model of neural networks based on LMA and using CUDA architecture was presented to improve the training performance. The use of this algorithm on graphics cards is new and presented excellent results, even with only one part of the training process running on the GPU.

The existing studies are limited to the use of CUDA architecture on the signals propagation through neural networks or training with simple gradient on MLP networks. The model introduced by this paper can train more types of artificial neural networks and in a more efficient way. Another important concern of this paper was the comparative experiments performed. In these experiments, all measures of time made on the CPU were made directly using MKL library from a program written in C language. This library has an exceptional performance and some comparisons made with it showed gains in the order of hundreds of times for some matrix calculations. This further enhances the results achieved since the proposed model was almost 30 times faster than the sequential model using MKL.

In future works, the whole training process will be transferred to the graphics card and the transfers of data made in each epoch will be replaced by an initial transfer added to other smaller one at the end, including the process output that comprises only the network weights vector. Thus, it's possible to run real-time training, which is required in various types of problems, for example, a system for up scaling video that needs to train a network on each frame.

References

1. Sheetal, L., Pinky, A., Narayanan, A.: High performance pattern recognition on gpu (2008)
2. Bakhoda, A., George, L., Fung, W., Wong, H., Aamodt, T.: Analyzing cuda workloads using a detailed gpu simulator. In: IEEE International Symposium on Performance Analysis of Systems and Software (2009)

3. Muniz, C., Figueiredo, K., Vellasco, M., Chavez, G., Pacheco, M.: Irregularity detection on low tension electric installations by neural network ensembles. In: IJCNN 2009, Rio de Janeiro, pp. 2176–2182 (2009)
4. Hagan, M.T., Menhaj, M.B.: Training feedforward networks with the marquardt algorithm. IEEE Transactions on Neural Networks 5(6), 989–993 (1994)
5. Haykin, S.: Neural Networks and Learning Machines, 3rd edn. Prentice-Hall (2008)
6. Intel. Math kernel library from intel
7. Jang, H., Park, A., Jung, K.: Neural network implementation using cuda and openmp. In: DICTA 2008: Proceedings of the 2008 Digital Image Computing: Techniques and Applications, pp. 155–161. IEEE Computer Society, Washington, DC (2008)
8. NVIDIA. CUDA Programming Guide. NVIDIA, Santa Clara, 2.3.1 edition (2009)