2017 IEEE International Parallel and Distributed Processing Symposium Workshops

Training Many Neural Networks in Parallel via Back-Propagation

Javier A. Cruz-López, Vincent Boyer
Graduate Program in Systems Engineering
Universidad Autónoma de Nuevo León
66451, Monterrey, Mexico
javier.cruz.003@gmail.com, vincent.boyer@uanl.edu.mx,

Didier El-Baz
LAAS-CNRS, Université de Toulouse, CNRS
Toulouse, France
elbaz@laas.fr

***Abstract*—This paper presents two parallel implementations of the Back-propagation algorithm, a widely used approach for Artificial Neural Networks (ANNs) training. These implementations permit one to increase the number of ANNs trained simultaneously taking advantage of the thread-level massive parallelism of GPUs and multi-core architecture of modern CPUs, respectively. Computational experiments are carried out with time series taken from the product demand of a Mexican brewery company; the goal is to optimize delivery of products. We consider also time series of the M3-competition benchmark. The results obtained show the benefits of training several ANNs in parallel compared to other forecasting methods used in the competition. Indeed, training several ANNs in parallel yields to a better fitting of the weights of the network and allows to train in a short time many ANNs for different time series.**

***Keywords*-Product Demand Forecasting, Neural Networks, Back-Propagation, GPU, Multiprocessing.**

I. Introduction

In recent years, artificial neural networks (ANN) have proven to be a powerful tool for classification and pattern recognition. One of the main reasons is its ability to learn from experience and from general information. The ANNs have been used in a wide variety of fields such as science, business and industry.

Back-propagation is an algorithm that has been widely used in training neural networks for its simplicity of implementation and its efficiency. This paper deals with the study of the implementation of the Back-propagation algorithm on a Graphics Processing Unit (GPU) and on a multi-core CPU such that several ANNs can be trained simultaneously. The objective is to increase the diversity of the search to obtain the best configuration for the problem under study. In particular, the expected benefits of training several ANNs in parallel compared to other forecasting methods are a better fitting of the weights of the network and quick training of many ANNs for different time series.

GPU are powerful graphics engines but also highly parallel computing accelerators, this characteristic has spawned the research community to map different computationally complex and demanding problems to the GPU. A parallel implementation via CUDA of the dynamic programming method for solving the knapsack problem on NVIDIA GPU is presented in [3] showing a speedup for large size instances compared with the sequential implementation. In [2] a survey with recent advances on GPU computing in Operation Research is presented, which shows that significant works have been proposed to parallelize meta-heuristics on such an architecture. This effort in general-purpose computing on the GPU has positioned it as a compelling alternative to traditional microprocessors in high-performance computing (HPC) systems.

In this paper, we detail the parallel implementation of the Back-propagation algorithm on GPU and multi-core processors. In particular, we study the benefits of training several ANNs in parallel. We display and analyze computational results for an industrial application (we consider time series taken from the product demand of a Mexican brewery company) and for problems of the M3 competition benchmark.

The paper is organized as follows. Section II is dedicated to the Back-propagation algorithm. In Section III, GPUs and their programming environment are presented. Previous works related to parallel ANNs are described in Section IV. Section V deals with our proposed GPU parallel implementation of the back-propagation algorithm. Computational experiments are presented and analyzed in Section VI. Finally, the conclusion and the future work are given in Section VII.

II. ANN and Back-propagation Algorithm

A. ANN

The first neural network model of ANN was proposed in 1943 by McCulloch and Pitts [12] in terms of a computational model of nervous activity. Neural Networks were originally an abstract simulation of biological nervous systems consisting of a set of units called neurons connected to each other (see [1]). A biological neuron receives inputs from other sources, combines them in some way, performs a generally nonlinear operation on the result, and then outputs the final result. It consists of dendrites, which receive input signals from other neurons; the soma, which processes these incoming signals over time, then turns the processed value into an output; and the axon, which carries the output from the neuron to other neuron dendrites (see [6] and [18]). The neuron collects the signals from their dendrites by summing the excitatory and inhibitory influences. If positive excitatory influences dominate, then the neuron produces a positive signal and sends this message to other neurons through the axoms. The neuron acts as a simple step function.

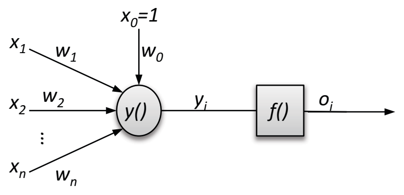

Fig. 1. Neuron scheme

The Figure 1 represents the scheme of a neuron. The formal neuron has n inputs $x_1, x_2, ..., x_n$ that model the signals coming from the dendrites. The inputs are labeled with the corresponding synaptic weights $w_1, w_2, ..., w_n$ that measure their permeabilities. Some of these synaptic weights may be negative to express their inhibitory character. A neuron could be biased, that is to say it has additional input x_0 with a constant value. Bias value allows the neuron to shift the activation function to the left or right, which may be critical for successful learning. The mathematical formulation of the neuron function is given by the weighted sum of input values as shown in equation (1):

$$y_i = \sum_{j \in J} w_j x_j, J = \{0, 1, 2, ..., n\} \tag{1}$$

where:

- J is the set of nodes in the input layer;
- x_j is the j-th input value; and
- w_j is the j-th weight from input layer to neuron y_i connected to input j.

An activation function (see equation (2)) is then applied in order to produce an output o_i. This function can be a step function, or a nonlinear activation function. The choice of the activation function depends on the problem to solve. However, nonlinear activation functions generally allow such networks to deal with nontrivial problems using only a small number of neurons.

$$o_i = f(y_i) \tag{2}$$

ANNs are powerful tools for pattern classification and recognition since they are able to learn and generalize from experience. They have been used for a wide variety of tasks in many different fields of industry and science [16], [25]. ANNs have become an important method for time series forecasting and have been found to be a contender when compared to traditional statistical time series models as shown in [26]. The ability to learn from experience is very useful for many practical problems since it is often easier to have data than to have good theoretical guesses about the systems from which data are generated. ANNs can infer the unseen part of the data even if the sample data contains noisy information.

B. Back-Propagation Algorithm

There are plenty of algorithms for training an ANN, such as back-propagation, conjugate gradient, cascade correlation, Levenberg-Marquardt, among others. Detailed information about those algorithms can be found in [4], [10], [18]. Back-propagation algorithm is one of the most used. This is due to its ease of programming and its enormous power to manipulate large amounts of data. This algorithm was popularized by Rumelhart et al. (1986) [19]. The advantage of back-propagation is the demonstration that layered networks using differentiable models could perform nontrivial calculations and offer attractive features such as fast response, learn from examples, and the ability to generalize beyond the training data. A typical network is composed by an input layer, an output layer, and at least one hidden layer. The number of hidden layer depends on the complexity of the problem [13]. Each layer is fully connected to the succeeding layer, as shown in Figure 2.

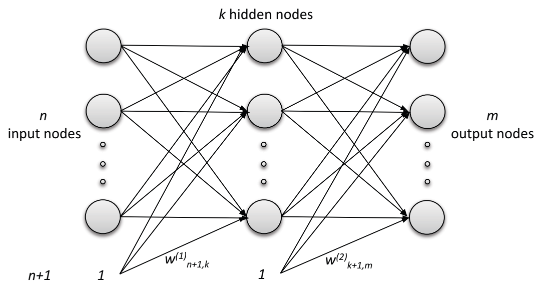

Fig. 2. Notation for the three-layered network

The first part in the back-propagation algorithm is the feed-forward section, which calculates the output of the network. Teaching process for a feed-forward network normally uses some variant of the Delta Rule, which starts with computing the difference between the current outputs and the desired outputs. In the sequel, we use the following notation in order to describe the multilayer neural network architecture.

- T_i is the target value i;
- $x_j^{(i)}$ is the j-th input of layer i;
- $o_j^{(i)}$ is the j-th output of layer i;
- $w_{j,k}^{(i)}$ is the j-th weight from layer i to neuron $x_j^{(i)}$ connected to neuron $x_k^{(i+1)}$; and
- $\delta_j^{(i)}$ is the error of output j of layer i.

The back-propagation algorithm works as follows. The network is first initialized by setting up all its weights to be small random numbers. The input pattern is then applied and the output calculated (the forward pass). The output of each neuron is calculated as described below, where equation (3) is the weighted sum of input values and (3) is the sigmoid activation function.

$$x_k^{(i)} = \sum_{j \in J} x_j^{(i-1)} w_{j,k}^{(i-1)} o_k^{(i)} = \frac{1}{1 + e^{-(x_k^{(i)})}} \tag{3}$$

The error of each output neuron is calculated according to equation (4), which is essentially the target value minus the output one ($T_j - o_j^{(i)}$), but as the sigmoid function is used the error is multiplied by the derivative sigmoid function.

$$\delta_j^{(i)} = o_j^{(i)}(1 - o_j^{(i)})(T_j - o_j^{(i)}) \tag{4}$$

This error is used to change the weights in such a way that the error will get smaller. Let $w_{j,k}^{*(i)}$ be the new (trained) weight and $w_{j,k}^{(i)}$ be the initial weight, then:

$$w_{j,k}^{*(i)} = w_{j,k}^{(i)} + \delta_k^{(i)} o_j^{(i)} \tag{5}$$

The errors for the hidden layer neurons is computed by back-propagating the errors from the output layer as shown in equation (6).

$$\delta_j^{(i)} = o_j^{(i)}(1 - o_j^{(i)})(\sum_{k \in K} \delta_k^{(i+1)} w_{j,k}^{(i+1)}) \tag{6}$$

Having obtained the error for the hidden layer neurons, the weights are changed according to equation (5). and the process is repeated until the error is minimal or after a fixed number of iterations. The pseudo code of the back-propagation algorithm is given in Algorithm 1.

Algorithm 1 The classic backpropagation algorithm

```
Require: ProblemSize, TrainingPatterns, maxIterations
1: weights ← InitializeWeights()
2: for $i = 1$ to maxIterarions do
3:     error = 0
4:     for $j =$ to numPatterns do
5:         pattern ← SelectPattern(TrainingPatterns)
6:         output ← ForwardPropagate(pattern, weights)
7:         BackwardPropagate(pattern, output, weights)
8:         error + = MSE(patterns, outputs)
9:     end for
10:    averageError = error/numPatterns
11: end for
```

The function **InitializeWeights**() initialize the weights with random values. The function **ForwardPropagate** (see Algorithm 2) computes the values of the outputs for each neuron from the input layer to the output one, based on equations (3) and (3). Then **BackwardPropagate** (see Algorithm 3) the error is backpropagated as in equations (4) and (6). Then, the function **UpdateWeights** (see Algorithm 4) updates the weights of the network using equation (5). After each iteration, the function **MSE** is called, where the Mean Square Error (MSE) per each pattern is computed, and finally the average error for all patterns is calculated.

III. General-Purpose Computing on GPU

General-Purpose Computing on Graphics Processing Units (GPGPU) is the use of a GPU (which typically handles computation for computer graphics) to perform computation

Algorithm 2 ForwardPropagate function

```
Require: input, weight, maxNumLayers, layerSize
1: $x$ ← input, $o$ ← output, $w$ ← weight, $M$ ← maxNumLayers
2: for $i = 1$ to $i =$ layerSize[1] do
3:     $o_i^{(1)} = x_j^{(1)}$
4: end for
5: for $i = 2$ to $i = M$ do
6:     for $j = 1$ to $j =$ layerSize[$i$] do
7:         sum = 0
8:         for $k = 1$ to $k =$ layerSize[$i - 1$] do
9:             sum = sum+($o_k^{(i-1)} w_{j,k}^{(i)}$)
10:        end for
11:        $o_j^{(i)} = 1/(1 + e^{-(sum)})$
12:    end for
13: end for
```

Algorithm 3 BackwardPropagate function

```
Require: target, weight, maxNumLayers, layerSize
1: $x$ ← input, $o$ ← output, $t$ ← target, $w$ ← weight, $M$ ← maxNumLayers
2: for $i = 1$ to $i =$ layerSize[$M$] do
3:     $\delta_i^{(M-1)} = o_i^{(M-1)}(1 - o_i^{(M-1)})(t_i - o_i^{(M-1)})$
4: end for
5: for $i = (M$-1) to $i = 1$ do
6:     for $j = 1$ to $j =$layerSize[$i$] do
7:         sum = 0
8:         for $k = 1$ to $k =$ layerSize[$i + 1$] do
9:             sum = sum+($\delta_k^{(i+1)} w_{k,j}^{(i+1)}$)
10:        end for
11:        $\delta_j^{(i)} = o_j^{(i)}(1 - o_j^{(i)})*$sum
12:    end for
13: end for
14: UpdateWeights(output, weights)
```

Algorithm 4 UpdateWeights function

```
Require: output, weight, maxNumLayers, layerSize
1: $o$ ← output, $w$ ← weight, $M$ ← maxNumLayers
2: for $i = 1$ to $M$ do
3:     for $j = 1$ to layerSize[$i$] do
4:         for $k = 1$ to layerSize[$i - 1$] do
5:             $w_{j,k}^{(i)} = w_{j,k}^{(i)} + (\delta_k^{(i)} o_j^{(i)})$
6:         end for
7:     end for
8: end for
```

in applications traditionally handled by the CPU. Microprocessors based on a single Central Processing Unit (CPU) drove rapid performance increases in computer applications for more than two decades. However, it has slowed since 2003, as shown in [10], due to energy consumption and heat dissipation issues that limited the level of productive activities that can be performed during each clock period within a single CPU. Microprocessor vendors have switched to models where multiple processing units are used in each chip to increase the processing power.

Highly parallel devices like GPUs have led the race of floating-point performance since 2003 [17]. The widespread adoption of GPUs in desktops and workstations has made them attractive as computing accelerators for high-performance parallel computing. Now modern GPUs are fully programmable, highly parallel architectures that deliver high throughput and hence can be used very efficiently for a variety of general purpose applications.

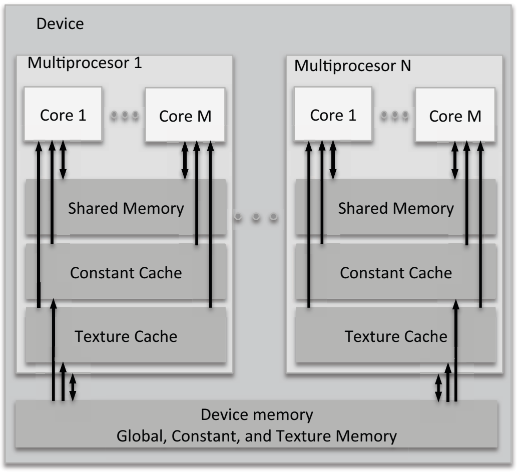

Fig. 3. The CUDA memory model

A. NVIDIA's GPU Architecture

NVIDIA's GPUs are very powerful and highly parallel computing architectures. GPUs have thousands of Compute Unified Device Architecture, CUDA cores and thousands of threads running concurrently on these cores [5]. Hence all kinds of computations for which many threads have to execute the same instruction concurrently are well-suited to run on GPU. All multiprocessors access a large global device memory for both gather and scatter operations. The memory model is displayed in Figure 3.

Shared memory access is faster than global memory access and usually slower than registers access. The shared memory is local to each multiprocessor, unlike device memory, and allows more efficient local synchronization. Each thread block within the multiprocessor accesses its own part of shared memory and this part of shared memory is not accessible by any other thread block of this multiprocessor or of other multiprocessors [20].

In order to define the threads a grid structure is used. The Grid consists of thread blocks. Each thread block is further divided into threads, which will run on the multi-processors. Figure 4 describes a two-dimensional grid structure and a two-dimensional block structure. Within a thread block, threads are organized together in warps, normally composed by 32 threads. All threads of a warp are scheduled together for execution.

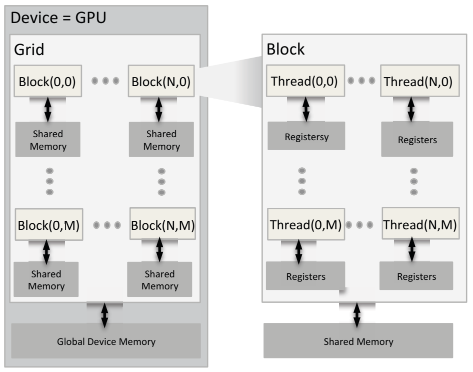

Fig. 4. The CUDA Grid Structure and Block Structure

B. NVIDIA GPU Programming Model

NVIDIA has designed a parallel computing platform and programming model based on C language called Compute Unified Device Architecture (CUDA), to use the massively parallel nature of GPU. CUDA contains a special C function called kernel, which launch the execution of the program on the graphics card on a fixed number of threads concurrently. CUDA Single Instruction Multiple Thread (SIMT) programming paradigm is a combination of serial and parallel executions. Figure 5 shows an example of this heterogeneous type of programming. The C code runs serially on CPU also called the host. Parallel execution is expressed by the kernel function which is executed on a set of threads in parallel on the GPU.

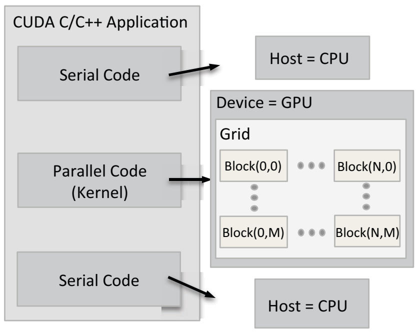

Fig. 5. The heterogeneous programming model of CUDA

The kernel function can only be invoked by serial code from the CPU. When the kernel function is called, the number of threads in a thread block and the number of threads within a grid must be specified. Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block, registers, a per-thread private memory, inputs, and output results.

IV. Related work

Several works on multiprocessing and ANNs algorithms have been published showing the advantage of using parallel architectures. A distributed memory-multiprocessor system is used in [24] to simulate a fully connected multi-layer neural network on back-propagation algorithm; this model is partitioned into subnetworks and each network is mapped on a processor; in this study the principal issue is how to partition the data. In [22] two different implementations of back-propagation via OpenMP are studied, one of them consists in partitioning the hidden layers among processors while the other divides the inputs among the processor maintaining a complete copy of the network on each processor. An implementation with GPU is also described in [15] where the training of the ANN is represented as a matrix multiplication. A partitioning scheme for multilayer network with back-propagation is presented in [21]. The partitioned network is mapped into a network of workstations to accelerate the training process.

An algorithm for image processing and pattern recognition is implemented in [7] using CUDA and OpenMP, showing an increment around 15 times faster than the implementations using CPU; in this CUDA implementation the NVIDIA CUDA Basic Linear Algebra Subroutines (CUBLAS) is used to set and solve the network matrix for the training stage. In [9] a Massively Parallel Computing System called SpiNNaker is described. It is characterized by massive processing parallelism and a high degree of interconnection among the processing units. The objective is to mimic neural computation, and to be able to simulate neural networks consisting of 10^9 neurons. The authors show that a similarly sized collection of biological neurons would run at the same speed.

A zero-order Takagi-Sugeno-Kang (TSK)-type Fuzzy Neural Network (FNN) algorithm is developed in [8] using the GPU architecture, which significantly reduce the training time as compared with a CPU implementation. In [14] a parallel implementation of a Spiking Neuronal Network is proposed using the CUDA framework, for increasing the training speed of ANNs. Locally-connected Neural Pyramid (LCNP) is a neural network model developed in [23]; this model is optimized for large-scale, high-performance object recognition and it is implemented using NVIDIA CUDA.

As seen in this section, all works are focused on creating massive parallel architecture as the size of neural networks is typically conceived as being very large. The ability to simulate them is generally limited by the speed and storage capacity of digital computers. However, there are situations for which size of data sets that we want to train is not very large, but there is a large number of different sets that need to be trained independently, that is the problem we are dealing with. As an example, a local company provided us some of its data sets which have 250 distribution centers and each of them has around 80 different items with different demand. This gives us a total of 20000 different data sets that need to be forecasted. The objectives of this GPU implementation is first, to train in parallel the same network among all cores in order to explore more solutions; secondly, to show the benefits of training different data sets in parallel.

V. Back-propagation Implementation on GPU

As mentioned in section III, programming on GPU is different from programming on traditional architectures. Amongst the main concerns are data management and memory bandwidth that impact directly on the performance of the algorithm. In this section we show how the back-propagation algorithm has been implemented on GPU.

In our implementation, each thread deals with an ANN. All variables containing the patterns and the initial weights of the network are copied to the register memory, which is private and has faster access than global memory. The pseudo-code described in Algorithm 5 shows how the algorithm is initiated. This initialization function requires the patterns to train the networks; the number of threads to execute; and the number of weights per network.

Algorithm 5 Network initialization

```
Require: patterns, numThreads, numWeights, numPatterns
1: allocMemoryGPU(patternsGPU, numPatterns)
2: allocMemoryGPU(weightsGPU,
   numThreads*numWeights)
3: fillRandom(weightsGPU)
4: patternsGPU ← patterns
5: output ← kernelTrainingANN << numThreads >>
   (patternsGPU, weightsGPU)
```

The function **allocMemoryGPU** allocates space on the global memory of the GPU for the variables; **fillRandom** generates random numbers and assigns them to *weightsGPU* which contains the weights for all the networks, i.e. the number of weights in a network per the total number of networks that will be trained in parallel; next, the patterns are copied from the CPU to the GPU global memory; finally the function **kernelTrainingANN** is called, this is the kernel function where back-propagation is executed (see details in Algorithm 6).

Algorithm 6 is similar to Algorithm 1. The function **kernelTrainingANN**, which runs on the GPU, copies the weights from the global memory to a new variable stored in the register memory.When the kernel finishes, the network with the lowest MSE is returned.

VI. Experimentation

The results presented in this section were obtained with a workstation HP z420 and processor Intel Xeon E5-1620 v2 3.70GHz with 4 Cores, and a GPU accelerator NVIDIA Tesla K20c with 2496 CUDA cores. The configuration used

Algorithm 6 Training the network

```
Require: patternsGPU, weightsGPU
outputs, averageError
weights ← copyWeights(weightsGPU)
for $i = 1$ to maxIteration do
    error = 0
    for $j = 1$ to numPatterns do
        pattern ← SelectPattern(patternsGPU)
        output ← ForwardPropagate(pattern, weights)
        BackwardPropagate(pattern, output, weights)
        error += EQM(patterns, outputs)
    end for
    averageError = error/numPatterns
end for
weightsGPU ← copyWeights(weights)
```

in GPU to execute the parallel code was 2480 blocks running 64 threads each of them giving a total of 158720 threads.

The forecast of 73 time series that correspond to product demand in a Mexican brewery company was carried out. These instances consist of 52 observations over one year. We train the artificial neural network using the first 34 observations and the remaining ones are kept to validate the forecast of the time series. Currently, the company uses several exponential smoothing methods for the demand forecast taking into account three years of information.

The neural network is composed of three layers: input, hidden, and output layers, where each layer has three neurons. We purposely use a small ANN in order to show that substantial results can be obtained even with such a network and to better analyze the effect of the approaches on the quality of the forecast.

The experiment has been carried out with the K20 GPU, once with all instances, i.e. the ANNs for the 73 products are trained in one kernel call. Indeed, 158702 threads are created and they are divided in 2174 networks per product. The execution time is around 54 seconds and when the program finishes it returns the network with the lowest MSE for each product demand. We also carried out experiments with the OpenMP version setting the time limit to 54 seconds. We use the Symmetric MAPE (SMAPE) in order to evaluate the results because the time series have some values equal to 0 for which the MAPE does not work. SMAPE is given in equation (7).

$$SMAPE = \frac{1}{n}\sum_{t=1}^{n}\frac{|F_t - A_t|}{(|A_t| + |F_t|)/2} \tag{7}$$

where A_t is the current value and F_t is the forecast value.

Table I shows the number of times and the associated percentage, respectively, the GPU and the OpenMP implementations obtain a better SMAPE in the forecast stage than the method used by the brewery company for the total demand of the 73 products. Table I also gives the number of times the company's algorithm performs better thant the GPU implementation. We observe that although our implementation uses less observations than the method from the brewery company, it yields to a better approximation of the demand for around 60% of the instances tested.

TABLE I
SMAPE OBTAINED DURING FORECAST

	GPU	OpenMP	Company
#	44	41	29
%	60.27	56.16	39.73

We have also considered time series provided by the M3-Competition, e.g., see http://forecasters.org/resources/time-series-data/m3-competition

The M3-Competition permits one to compare the performance of time series forecast methods, some of which have been designed by recognized experts [11]. In our experimentation, we select 146 instances of the M3-competition corresponding to micro economy data. They consist in 20 yearly observations where 14 observations are taken to train the neural networks and the remaining 6 observations are used to test their accuracy.

We compare three different back-propagation algorithms. The first case corresponds to a sequential algorithm, i.e., the classic back-propagation method described in Section II. The second method is a OpenMP shared memory multiprocessing algorithm; in this particular case, parallel computing is performed on a multi-core CPU. The third algorithm is carried out on GPU, see Section V.

A time limit for the sequential and OpenMP implementations is imposed, which corresponds to the processing time required by the GPU implementation. Table II shows the time needed to run the GPU version, and the number of ANNs trained in that time by the different implementations. Each neural network training carries out one million iterations and the ANN with the best MSE is returned. As can be seen in Table II, the OpenMP algorithm can explore four times more solutions than the Sequential algorithm. The GPU parallel algorithm can explore **496** times more solutions than the OpenMP algorithm, and **2034** times more solutions than the Sequential algorithm.

Table III shows the number of instances where the best MSE is obtained for the different implementations. Note that for all instances the GPU implementation obtains the lowest error. The MSE is the error computed over all the patterns used for training the ANN but it does not show the accuracy of the forecast.

TABLE III
INSTANCES WITH THE LOWEST MSE OBTAINED

	GPU	OpenMP	Sequential	Total
#	146	0	0	146
%	100	0	0	100

During the training and the forecast stage, the Mean Ab-

TABLE II
NUMBER OF ANNS TRAINED

	GPU	OpenMP	Sequential	Processing Time
#Networks	158720	320	78	$56s$

solute Percentage Error (MAPE) is calculated as shown in equation (8).

$$MAPE = \frac{1}{n}\sum_{t=1}^{n}\frac{A_t - F_t}{A_t} \tag{8}$$

where A_t is the current value and F_t is the forecast value.

Table IV shows that in most cases the ANN obtained by the GPU implementation has a lower MAPE with respect to the others implementations (during the training phase). In contrast, one can see in Table V that the sequential algorithm obtains better forecasting performance in more instances tested than the GPU and OpenMP implementations. This can be explained by an overfitting of the weights.

TABLE IV
MAPE OBTAINED DURING THE TRAINING STAGE

	GPU	OpenMP	Sequential	Total
#	83	33	30	146
%	56.849	22.603	20.548	100

TABLE V
MAPE OBTAINED DURING FORECAST

	GPU	OpenMP	Sequential	Total
#	53	33	60	146
%	36.301	22.603	41.096	100

To analyze in detail these results, Table VI displays the percentage difference between the MSE obtained with the different algorithms. We note that for almost 70% of the instances, this difference is below 10%. However, considering that the best MSE obtained with each implementation is of the order of 10^{-4}, a difference below 10% could be considered as negligible.

Table VII shows the average of the MAPE obtained with 25 different forecast methods published by the M3-Competition and our approaches for the 146 instances used for experimentation. The three proposed implementations achieve in most cases a lower error compared to the average one. Besides, we note that the OpenMP algorithm obtains in average the best results compared to the sequential and the GPU implementations. It seems to offer the best compromise between the number of ANNs explored during the training phase and weights fitting .

VII. CONCLUSION AND FUTURE WORK

In this paper, we compare parallel back-propagation algorithms for training multiple Artificial Neural Networks simultaneously; we consider their implementations on GPU computing accelerators via CUDA and multi-core CPU via OpenMP.

We carry out computational tests on real data issued from the product demand in a brewery company and instances taken from the literature. The results show that the parallel implementations are able to explore a greater number of solutions and to obtain artificial neural network with a low Mean Square Error. The proposed parallel implementations have a competitive Mean Absolute Percentage Error as compared with other methods presented in the M3-Competition benchmark. Besides, with the real data, we were able to improve the forecast for 60% of the products and all the artificial neural networks were trained in less than one minute.

The great advantage of the parallel implementations is the number of different ANNs that can be trained in the same amount of time. It is a good alternative when many different time series have to be forecasted. For instance, the GPU implementation is able to train up to 159744 ANN simultaneously. However, we note that weights overfitting can occur in the training phase. A balance should be found between the number of ANNs trained and the performance of the best ANN reported. Hence, training a single ANN is not a good strategy, it is indeed better to trained in parallel different ANNs corresponding to different forecasts. This approach allows to decrease the number of threads dedicated to a particular instance and to improve the efficiency of the training phase.

Future work concerns tuning the Mean Square Error; indeed, the risk of having an overfitting problem tends to increase when the Mean Square Error is close to 0. We shall consider also the design of a parallel multi-configuration algorithm that will evaluate a time series with different configurations of the artificial neural network, e.g. different number of layers and neurons. The objective is to obtain the ANN with the smallest error and with the best configuration.

AKNOWLEDGMENTS

Javier Cruz and Vincent Boyer are very grateful to CONACYT for support through a full time scholarship. Didier El Baz thanks NVIDIA corporation.

REFERENCES

[1] D. Anderson and G. McNeill, *Artificial neural networks technology*, Kaman Sciences Corporation, vol. 258, pp. 302–462, 1992

[2] V. Boyer and D. El Baz, *Recent advances on GPU computing in Operations Research*, 27th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW 2013), pp. 1778–1787, 2013.

[3] V. Boyer and D. El Baz, Didier and M. Elkihel, *Solving knapsack problems on GPU*, Computers & Operations Research, vol. 39, N 1, pp. 42–47, Elsevier, 2012.

TABLE VI
PERCENTAGE-DIFFERENCE AMONG IMPLEMENTATIONS

Algorithm		Percentage-difference	#Instance	%Instance
GPU	Sequential	$< 10\%$	98	67.123
		$< 5\%$	63	43.151
GPU	OpenMP	$< 10\%$	120	82.192
		$< 5\%$	89	60.959
OpenMP	Sequential	$< 10\%$	130	89.041
		$< 5\%$	112	76.712

TABLE VII
AVERAGE MAPE

Method	Forecast horizon						Average	
	1	2	3	4	5	6	1 to 4	1 to 6
NAIVE2	11.35	29.01	34.40	34.52	35.59	35.87	27.32	30.12
SINGLE	11.60	29.10	34.22	34.31	35.34	35.65	27.31	30.04
HOLT	12.61	31.07	41.50	45.28	50.17	52.80	32.62	38.91
DAMPEN	11.50	29.50	36.70	39.12	43.33	44.82	29.21	34.16
WINTER	12.61	31.07	41.50	45.28	50.17	52.80	32.62	38.91
COMB S-H-D	11.14	28.93	36.33	38.21	41.45	42.77	28.65	33.14
B-J auto	12.57	30.31	36.47	38.56	40.42	41.61	29.48	33.33
AutoBox1	13.82	33.17	40.47	43.27	46.15	49.46	32.68	37.72
AutoBox2	11.02	28.03	**32.77**	33.85	**34.93**	35.79	**26.42**	**29.40**
AutoBox3	13.98	32.17	37.90	37.68	39.34	40.42	30.43	33.58
ROBUST-Trend	**10.29**	28.56	35.61	36.30	38.13	39.23	27.69	31.35
ARARMA	13.73	32.91	41.33	45.12	50.44	55.53	33.27	39.84
AutoANN	13.42	31.36	36.22	37.18	37.33	37.38	29.54	32.15
Flors-Pearc1	11.79	29.41	36.43	38.00	40.36	42.07	28.91	33.01
Flors-Pearc2	12.74	29.91	34.28	34.10	35.08	35.52	27.76	30.27
PP-Autocast	11.44	29.44	36.51	38.85	42.98	44.43	29.06	33.94
ForecastPro	12.07	29.46	35.98	38.24	40.78	42.41	28.94	33.16
SMARTFCS	11.38	28.51	35.04	36.02	38.16	39.38	27.74	31.41
THETAsm	10.74	28.04	33.71	33.82	35.31	35.75	26.58	29.56
THETA	11.94	29.93	36.17	38.25	41.14	42.43	29.07	33.31
RBF	11.47	29.16	35.39	35.44	36.97	37.14	27.86	30.93
ForcX	11.89	28.83	34.01	34.55	36.10	36.91	27.32	30.38
GPU	15.82	27.99	33.67	34.78	36.07	35.89	28.06	30.70
OpenMP	15.56	**27.62**	33.14	**33.66**	35.10	35.41	27.50	30.08
Sequential	16.04	27.90	33.44	33.68	35.00	**35.40**	27.76	30.24
Average	12.50	29.66	36.13	37.52	39.83	41.08	28.95	32.79

[4] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd edition, Prentice Hall, Upper Saddle River, NJ, USA, ISBN: 0132733501, 1998.

[5] R. Inam, *An Introduction to GPGPU Programming-CUDA Architecture*, Mälardalen University, Mälardalen Real-Time Research Centre, 2011.

[6] F. Izaurieta and C. Saavedra, *Redes neuronales artificiales*, Departamento de Física, Universidad de Concepción, Concepción, Chile, 2000.

[7] H. Jang and A. Park and K. Jung, *Neural network implementation using CUDA and OPENMP*, Digital Image Computing: Techniques and Applications (DICTA], pp. 155–161, 2008.

[8] C.-F. Juang and T.-C. Chen and W.-Y. Cheng, *Speedup of implementing fuzzy neural networks with high-dimensional inputs through parallel processing on graphic processing units*, IEEE Transactions on Fuzzy Systems, Vol. 19, N. 4, pp. 717–728, IEEE, 2011.

[9] M. M. Khan and D. Lester and L. Plana and A. Rast and X. Jin and E. Painkras and S. Furber, *SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor*, Neural Networks, IEEE World Congress on Computational Intelligence, pp. 2849–2856, 2008.

[10] D. Kirk, and W. Wen-Mei, *Programming massively parallel processors: a hands-on approach*, Newnes, 2012.

[11] S. Makridakis and M. Hibon, *The M3-Competition: results, conclusions and implications*, International Journal of Forecasting, vol. 16, N. 4, pp. 451–476, 2000.

[12] W. McCulloch and W. Pitts, *A logical calculus of the ideas immanent in nervous activity*, BMB, vol. 5, N. 4, pp. 115–133, Springer, 1943.

[13] N. Murata and S. Yoshizawa and S. Amari, *Network information criterion-determining the number of hidden units for an artificial neural network model*, IEEE Transactions on Neural Networks, vol. 5, N. 6, pp. 865–872, 1994.

[14] T. Nowotny, *Parallel implementation of a spiking neuronal network model of unsupervised olfactory learning on NVIDIA CUDA* The 2010 International Joint Conference on Neural Networks (IJCNN), pp. 1–8, 2010.

[15] K.-S. Oh and K. Jung, *GPU implementation of neural networks*, Pattern Recognition, vol. 37, N. 6, pp. 1311–1314, 2004.

[16] A. Rehman and T. Saba, *Neural networks for document image preprocessing: state of the art*, AIR, vol. 42, N. 2, pp. 253–273, 2014.

[17] C. J. Reimúndez, *Estudio de rendimiento en GPU*, Master degree thesis, Universidad Complutense de Madrid, Spain, 2010.

[18] R. Rojas and J. Feldman, *Neural Networks: A Systematic Introduction*, Springer Berlin Heidelberg, 2013.

[19] D. E. Rumelhart and G. E. Hinton and R. J. Williams, *Learning internal representations by error propagation*, DTIC Document, 1985.

[20] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*, Addison-Wesley Professional, 2010.

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

[22] R. K. Thulasiram and R. Rahman, and P. Thulasiraman, *Neural network training algorithms on parallel architectures for finance applications*, International Conference on Parallel Processing and Workshops, vol. 236, 2003.

[23] R. Uetz and S. Behnke, *Large-scale object recognition with CUDA-accelerated hierarchical neural networks*, IEEE International Conference on Intelligent Computing and Intelligent Systems, 2009. ICIS 2009, vol. 1, pp. 536–541, 2009.

[24] H. Yoon and J. Nang and S. R. Maeng, *Parallel simulation of multilayered neural networks on distributed-memory multiprocessors*, Microprocessing and Microprogramming, vol. 29, N. 3, pp. 185-195, 1990.

[25] G. P. Zhang, and E. Patuwo and M. Hu, *Forecasting with artificial neural networks: The state of the art*, IJF, vol. 14, N. 1, pp. 35-62, 1998.

[26] G. P. Zhang, *Neural networks for time-series forecasting*, Handbook of Natural Computing, pp. 461-477, 2012.