Facial Recognition Using Neural Networks over GPGPU

Juan Pablo Balarini, Martín Rodríguez, and Sergio Nesmachnow

Centro de Cálculo, Facultad de Ingeniería
Universidad de la República, Uruguay
{jbala87, martinr87}@gmail.com, sergion@fing.edu.uy

Abstract. This article introduces a parallel neural network approach implemented over Graphic Processing Units (GPU) to solve a facial recognition problem, which consists in deciding where the face of a person in a certain image is pointing. The proposed method uses the parallel capabilities of GPU in order to train and evaluate a neural network used to solve the abovementioned problem. The experimental evaluation demonstrates that a significant reduction on computing times can be obtained allowing solving large instances in reasonable time. Speedup greater than 8 is achieved when contrasted with a sequential implementation and classification rate superior to 85 % is also obtained.

Keywords: Face recognition, Neural Networks, Parallel Computing, GPGPU.

1 Introduction

Face recognition can be described as the ability to recognize people given some set of facial characteristics. Nowadays, it has become a popular area of research in computer vision and image analysis, mainly because we can find such recognition systems in objects of everyday life such as cellphones, security systems, laptops, PCs, etc. [21,22]. Another key element is that the high computing power now available makes these image recognition systems possible.

Using an image of a human face, an algorithm is proposed to evaluate and decide where that face is pointing. Each image is classified into one of four classes according to the direction where is facing (those classes are: left, right, up and straight).

For certain types of problems, artificial neural networks (ANN) have proven to be one of the most effective learning methods [2], built of complex webs of interconnected neurons, where each unit takes a number of real-valued inputs and produces a single real-valued output. Also the Backpropagation algorithm is the most commonly used ANN learning technique, which is appropriate for problems where the target function to be learned is defined over instances that can be described by a vector of predefined features (such as pixel values), also the target function output may be discrete-valued, real-valued, or a vector of several real or discrete-valued attributes. Additionally the training examples may contain errors, and fast evaluation of the learned function may be required. All this makes ANN a good option in image recognition problems. A survey of practical applications of ANNs can be found on [14].

One of the main inconvenient of ANNs is the time needed to perform the training phase, which generally is quite high for complex problems. As the number of hidden layers and neurons grows, the required time for the learning process of the ANN and for the evaluation of a new instance grows exponentially. On the other hand, the rate of successful classification of new instances increases as well. Note that generally, the more training examples the network is provided, the more effective it will be (and the longer it will take too). Therefore it is of special interest to perform training with a large number of neurons in the hidden layer and with a significant number of training examples, but with a relatively low training time.

It is interesting to note that every neuron in each layer can make their calculations independently of others in the same layer. This means that for any given layer, parallel computations can take place, and some parallel architecture could be used to take advantage of this.

Promising [20] work is being made in the area of general purpose GPU computing, principally in problems with parallel nature. GPU implementations allow obtaining significant reduction in the execution times of complex problems when compared with traditional sequential implementations on CPU [9]. Despite the fact GPUs were originally designed for the sole purpose of rendering computer graphics they have evolved into a general purpose computing platform with enough power and flexibility to make many computationally-intensive application perform better than on a CPU [12,13]. This can be explained by the significant disparity between CPUs and GPUs which rises every year. In this work, we propose an algorithm that takes advantage of this parallel architecture to obtain an algorithm that outperforms another that uses a sequential implementation.

This work focuses in the field of machine learning, high performance parallel computing, and using graphics processing units for general purpose computing, as it develops an algorithm that significantly improves the ANN training and classification time, as contrasted with a sequential algorithm.

The main contributions of this article are that a parallel face recognition algorithm is obtained that develops good classification rates in reasonable execution times, also this algorithm can be easily modified to recognize other features of a human face without changing those execution times. Furthermore, it demonstrates how the GPGPU platform is a very good option when it is desired to improve the execution time of a certain problem.

The rest of the paper is organized as follows. Section 2 introduces GPU computing and the CUDA programming model, section 3 presents a conceptual framework then, section 4 presents related work. Section 5 introduces the proposed solution and provides implementation details. In section 6 an experimental analysis is made and finally, section 7 presents the work conclusions and future work.

2 GPU Computing

GPUs were originally designed to exclusively perform the graphic processing in computers, allowing the Central Process Unit (CPU) to concentrate in the remaining computations. Nowadays, GPUs have a considerably large computing power, provided by hundreds of processing units with reasonable fast clock frequencies. In the last ten years, GPUs have been used as a powerful parallel hardware architecture to achieve efficiency in the execution of applications.

2.1 GPU Programming and CUDA

Ten years ago, when GPUs were first used to perform general-purpose computation, they were programmed using low-level mechanism such as the interruption services of the BIOS, or by using graphic APIs such as OpenGL and DirectX. Later, the programs for GPU were developed in assembly language for each card model, and they had very limited portability. So, high-level languages were developed to fully exploit the capabilities of the GPUs. In 2007, NVIDIA introduced CUDA [11], a software architecture for managing the GPU as a parallel computing device without requiring mapping the data and the computation into a graphic API.

CUDA is based in an extension of the C language, and it is available for graphic cards GeForce 8 Series and superior. Three software layers are used in CUDA to communicate with the GPU (see Fig. 1): a low-level hardware driver that performs the CPU-GPU data communications, a high-level API, and a set of libraries such as CUBLAS for linear algebra and CUFFT for Fourier transforms.

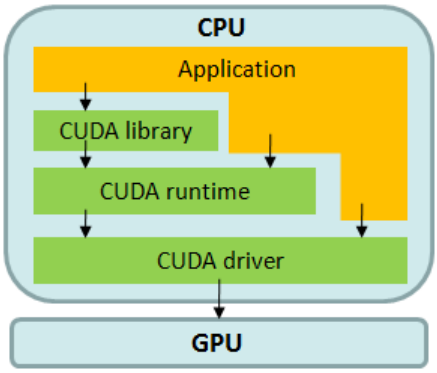

Fig. 1. CUDA *Architecture*

For the CUDA programmer, the GPU is a computing device able to execute a large number of threads in parallel. A procedure to be executed many times over different data can be isolated in a GPU-function using many execution threads. The function is compiled using a specific set of instructions and the resulting program (*kernel*) is loaded in the GPU. The GPU has its own DRAM, and the data are copied from it to the RAM of the host (and vice versa) using optimized calls to the CUDA API.

The CUDA architecture is built around a scalable array of multiprocessors, each one with eight scalar processors, one multithreading unit, and a shared memory chip. The multiprocessors are able to create, manage, and execute parallel threads, with reduced overhead. The threads are grouped in *blocks* (with up to 512 threads), which are executed in a single multiprocessor, and the blocks are grouped in *grids*. When a CUDA program calls a grid to be executed in the GPU, each one of the blocks in the grid is numbered and distributed to an available multiprocessor. The multiprocessor receives a block and splits the threads in *warps*, a set of 32 consecutive threads. Each warp executes a single instruction at a time, so the best efficiency is achieved when the 32 threads in the warp executes the same instruction. Each time that a block finishes its execution, a new block is assigned to the available multiprocessor.

The threads access the data using three memory spaces: a *shared memory* used by the threads in the block; the *local memory* of the thread; and the *global memory* of the GPU. Minimizing the access to the slower memory spaces (the local memory of the thread and the global memory of the GPU) is a very important feature to achieve efficiency. On the other side, the shared memory is placed within the GPU chip, thus it provides a faster way to store the data.

3 Face Recognition Using Artificial Neural Networks in GPU

3.1 Face Pointing Direction

The face pointing direction problem consists in recognizing where a human face is pointing (up, left, right and center) in a certain image. This problem has many practical applications such as detecting where a driver is looking while driving (raising an alarm if the driver fell asleep), a computer mouse for impaired people that moves according to head movements (i.e. face direction), a digital camera software that only takes a picture if all individuals are looking at the camera, etc. Traditional methods to solve this problem include ANNs [17, 2], evolutionary algorithms [15, 16], problem specific heuristics, etc., but in general, sequential implementations are used.

3.2 Artificial Neural Networks

ANNs provide a general practical method for learning real-valued, discrete-valued, and vector-valued functions from examples. Several algorithms (such as *backpropagation*) can be used to tune network parameters to best fit a training set of input-output pairs. ANNs are robust to errors in the training data and has been successfully applied to problems such as image recognition, speech recognition, and learning robot control strategies [2].

Figure 2 presents the general schema of an ANN. There is a set of *neurons* connected with each other. Each neuron receives several input data, perform a linear combination (result a) and then produces the result of the neuron, which is the evaluation of some function $f(x)$ for the value $x = a$.

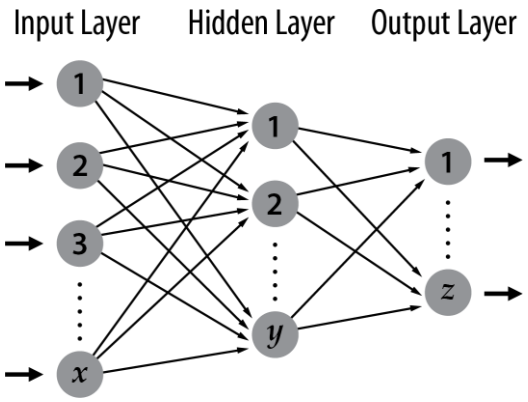

Fig. 2. Schema of an ANN.

The neurons are grouped in several layers:

- Input layer: receives the problem input
- Hidden layer/s: receives data from other neurons (typically from input layer or from another hidden layer), and forwards the processed data to the next layer. In an ANN, there may be multiple hidden layers with multiple neurons each.
- Output layer: this layer may contain multiple neurons and it determines the output of the processing for a certain problem instance.

Fig. 3 shows a schema for a neuron. First, a linear combination of the neuron input data (x_i, weights w_i, $i \in [1, n]$, and an independent coefficient w_0) is made. Then the output is evaluated at some well-known *activation function*, to produce the neuron output.

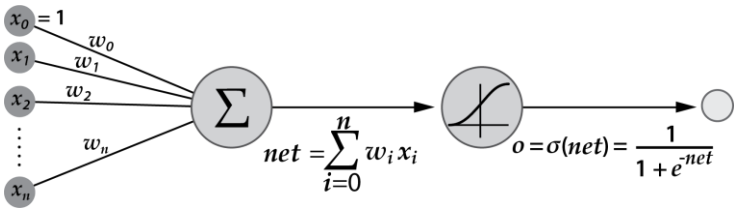

Fig. 3. Schema of a single neuron in an ANN.

3.3 Face Recognition Using GPGPU

In this article, an ANN is used to solve the face recognition problem, trained with the backpropagation algorithm. Backpropagation learns the weights for a multilayer network with a fixed set of units and interconnections, by applying the gradient descent method to minimize the squared error between the network output values and the target values for these outputs. The learning problem faced by backpropagation implies searching in a large space defined by all possible weight values for all neurons. The backpropagation method applied in this work (stochastic gradient descent version for feedforward networks) is described in Algorithm 1.

Backpropagation (training_examples, n, n_{in}, n_{out}, n_{hidden})

Each training example is a pair of the form$\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network of input values, and $\vec{t}$ is the vector of target network output values.

n is the learning rate, n_{in} is the number of network inputs, n_{hidden} the number of units in the hidden layer, and n_{out} the number of output units.

The input unit from unit i into unit j is denoted x_{ji} , and the weights from unit i to unit j is denoted w_{ji}.

- Create a feed-forward network with n_{in} inputs, n_{hidden} hidden units, and n_{out} output units.
- initialize all network weights to small random numbers
- while the termination condition is not met, do

  for each $\langle \vec{x}, \vec{t} \rangle$ in training_examples,do

  {propagate the input forward through the network}

  input the instance $\vec{x}$ to the network and compute the output o_u of every unit u in the network

  {propagate the errors backward through the network}

  for each network output unit k, calculate its error term δ_k

  $$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

  for each hidden unit h, calculate its error term δ_h

  $$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{kh} \delta_k$$

  update each network weight w_{ji}

  $$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}, \text{ where } \Delta w_{ji} = n\delta_j \, x_{ji}$$

Algorithm 1. Stochastic gradient descent version of the backpropagation algorithm for feed-forward networks.

Algorithm 1 begins by constructing a network with the desired number of hidden and output units and initializing all network weights to small random numbers. Given this fixed network structure, the main loop of the algorithm iterates over the training examples. For each training example, it applies the network to the example, computes the gradient with respect to the error on this example, and then updates all weights in the network. This gradient step is iterated (using the same training examples multiple times) until the network performs acceptably well [2]. For evaluating a single instance (not to train the network), only the propagation of the input data through the network is made. The presented ANN uses neurons of sigmoid type with activation function:

$$F(x) = \frac{1}{1+e^{-x}} \tag{1}$$

To solve the face recognition problem in GPU, a specific version of the backpropagation algorithm was implemented. The generic schema in Algorithm 1 was adapted to execute on a GPU, mainly taking into account the communication restrictions between the GPU processing units. Before calling a function that runs on GPU a function-dependent domain decomposition is applied. In general, certain GPU threads are assigned to execute over certain neurons on the ANN. The domain decompositions are always performed to maximize the parallel execution, (i.e. each GPU thread can work independent from each other), and to avoid serializations in the memory access. A detailed description of domain decomposition is presented in section 5.2.

Figure 4 presents a schema of the ANN training, showing that the train() function is a concatenation of functions that execute in parallel.

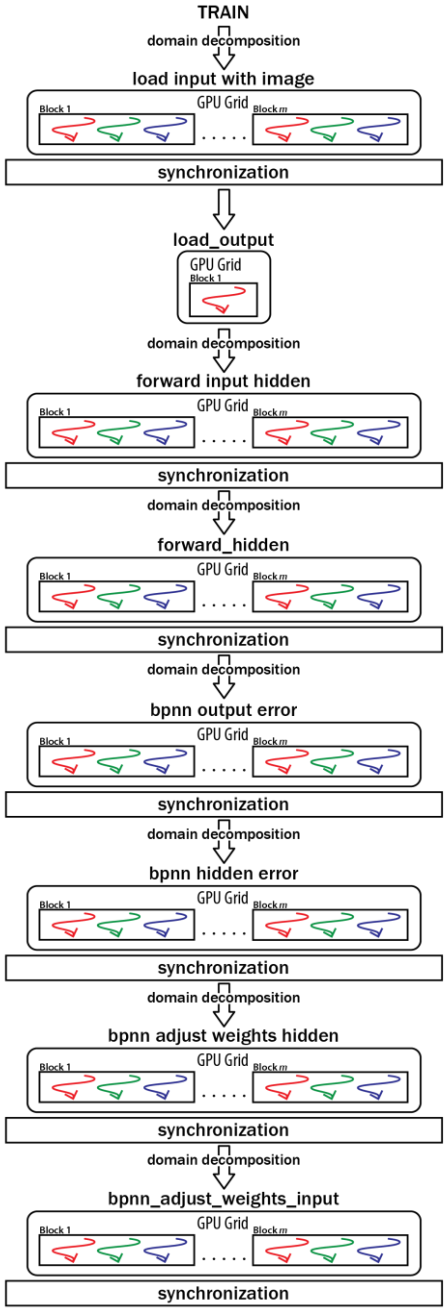

Fig. 4. ANN training: parallel approach.

4 Related Work

Lopes and Ribeiro [9] presented an analysis of an ANN implementation executing on GPU, showing how the training and classification times can be reduced significantly (ranging from 5× to 150×, depending on the complexity of the problem). They also conclude that the GPU scales better than the CPU when handling large datasets and complex problems. In this work the authors recognizes two sources of parallelism: the outputs of the neurons can be computed in parallel, and all the samples (patterns) on a dataset can be processed independently. The parallel ANN takes advantage of parallelism in the three training phases (forward, robust learning and backpropagation). Several problems were tackled in the experimental analysis, including solving $f(x) = \sin(x)/x$, and several classification and detection problems such as: the two-spirals problem, the sonar problem, the covertype problem, the poker hand problem, the ventricular arrhythmias problem and face recognition of the Yale face database [18], containing 165 gray scale faces images of 64×64 pixels of 15 individuals.

Jang et al. [6] introduced a parallel ANN implementation using CUDA applied to text recognition on images. Processing times up to 5 times faster than a CPU implementation were obtained. In this work, parallelism is achieved through computing in parallel all the linear combinations made when some neuron calculates their output, and also when the sigmoid function is computed on each neuron. In this case, text detection was performed over three image sizes (320×240, 571×785 and 1152×15466), always using 30 neurons on the hidden layer.

Solving a similar problem, Izotov et al. [7], used an ANN-based algorithm to recognize handwritten digits, using a CUDA-based implementation. The training time improvements were about 6 times less than an algorithm executed on CPU, and on instance classification there were reductions of about 9 times in execution time. This work represented some ANN features as matrixes and took advantage of the CUBLAS library (a linear algebra library over CUDA driver level) for calculating matrix multiplications (thus achieving parallelism) using 8-bit gray scale 28×28 pixel image instances of handwritten digits from the public domain MNIST database [19].

Nasse et al. [8] solved the problem of locating the direction of a human face in space, using ANN on a GPU. Parallelization of the solutions was achieved through dividing the image into several rectangles, and computing each one in parallel. This implementation obtains classification values up to 11 times faster than an implementation which runs on CPU, and was trained using 6000 non-faces and 6000 faces with three different sizes (378×278, 640×480 and 800×600 pixels).

Shufelt and Mitchell [4] solved a similar problem to the one proposed in this work (deciding whether an image is of a certain person or not) by using a sequential method. Their proposal was the starting point for the parallel solution presented here.

The analysis of the related work allows concluding that ANN implementations that execute over a GPU can obtain very good results when contrasted with a CPU-only implementation. Taking this fact into account, our purpose here is to develop an ANN to recognize certain features of a human face in a short period of time. If training time can be reduced considerably by using parallel GPU infrastructures, the solution will overcome one of the main disadvantages of traditional ANN implementations.

5 Implementation Details

The proposed parallel implementation applies the ideas in the sequential algorithm by Shufelt and Mitchell [4] for recognizing if a given picture is of a certain person. Thus, the method has to be slightly modified to obtain the expected solution for the face recognition problem. Moreover, in the parallel implementation, the possibility of working with a second layer of hidden neurons was included.

5.1 Software modules

The proposed solution uses five modules, which implement all functions needed by the algorithm. First, facetrain.cu contains the main method and is the module that performs the calls to the other functions. Then, backprop.cu implements the ANN and all the auxiliary functions needed to work with it. The proper interaction between the ANN and the images is solved in imagenet.cu. The pgmimage.cu library is used to work with images in pgm format. Finally, constants.h contains the entire configuration for the correct execution of the algorithm.

Training is a key element in the algorithm. The diagram in Fig. 5 explains the required steps needed to train the ANN for the entire trainset.

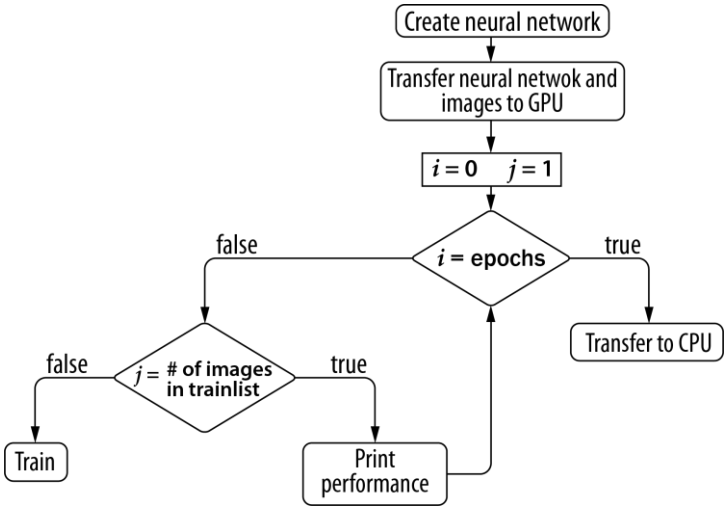

Fig. 5. ANN training: functional schema.

The GPU architecture is best exploited when performing training and classification. For example (see Fig. 6), when forwarding from input layer to hidden layer, the parallel algorithm creates as many blocks as neurons are in the hidden layer (each block will work with one neuron on the hidden layer), and each thread in a block will compute the linear combination of the weight that goes to that hidden neuron by the data that comes from the input layer (the algorithm works with the same number of threads per block as input neurons are). Similar levels of parallelism are achieved in the rest of the functions, obviously changing the role of each block and each thread in a block.

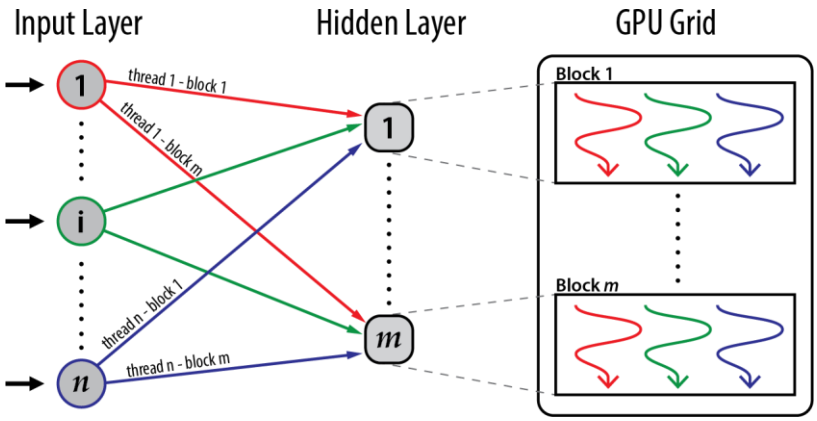

Fig. 6. Parallelism example: forwarding from input layer to hidden layer.

5.2 Tasks Executed on GPU

Key parameters such as the grid and block size used for the function invocations performed on GPU are of special interest for the overall performance of the algorithm. The following functions are called in both the training and evaluation tasks. The function *load_input_with_image()* is called with as many block as rows the image has, and as many threads per block as columns the image has. This function loads each image into the neurons of the input layer for later use (each block loads a row of the image). After that, *forward_input_hidden()* computes the outputs of the neurons in the hidden layer, using the data from the input layer. It is called with as many blocks as the number of hidden neurons in the network and as many threads per block as the GPU allows (1024 in our case). Each block computes the output of one neuron on the hidden layer, which is the linear combination performed by several threads (see Section 2). The function *forward_hidden()* works like the previous function, obtaining the output of the hidden layer. Next, *load_output()* loads the expected output of the ANN for a certain image. It is invoked with one block with one thread because of its simplicity.

The function *evaluate_performance()* computes the error of an image and checks if the output of the ANN matches the expected one. It is called with one block with one thread, due to the small processing performed. The functions *bpnn_output_error()* and *bpnn_hidden_error()* are used to compute the error in the output and hidden layer, respectively. The first one is called with one block and with as many threads per block as output neurons the ANN has. The second one with as many blocks as hidden neurons are and with as many threads per block as output neurons are. The function *bpnn_adjust_weights_hidden()* adjust the weights from the hidden layer to the output layer. It is called with as many blocks as output neurons are, and with as many threads per block as number of hidden neurons are plus one. For each block, it adjusts the weights that go from hidden neurons (as many as threads) to output neurons. Finally, *bpnn_adjust_weights_input()* adjusts weights that go from input layer to output layer. It works like the previous function, with the difference that the number of threads must be larger, due to the number of neurons the input layer has.

5.3 Other GPU considerations

Throughout the provided implementation, all constants have their type defined. This implementation decision was made because a precision loss was detected when performing conversions (e.g. double to float), affecting the numerical efficacy of the proposed algorithm. Since all GPU memory must be contiguous, static structures are used, because when transferring data from CPU to GPU the *cudaMemcpy* function copies only contiguous memory directions. Moreover, the CPU stack size had to be enlarged to 512 MB in order to allow storing 128×120 images or larger.

In addition, certain implementation decisions were taken to improve the final performance of the proposed algorithm. First, it was decided to use shared memory in certain GPU kernels, to hide the latency of global memory access. Another element to take into account is that most threads running on GPU performed several calculations, in order to avoid the limit for the number of threads per block in the execution platform (1024 in CUDA compute capabilities 2.0).

A weakness of the implementation is that heavily relies on the used hardware (especially with the compute capabilities of the graphics card). This will impact on the number of threads per block that can be created and in the use of certain CUDA functions (i.e. use of the *atomicAdd* function with data of float type).

6 Experimental Analysis

This section reports the results obtained when applying the parallel GPU implementations of the face pointing direction problem for a set of problem instances. A comparative analysis with a sequential implementation is performed, and the obtained speedups (the quotient between execution time in the sequential implementation and execution time in the parallel implementation) are also reported.

6.1 Development and Execution Platform

The GPU algorithm was developed on an AMD Athlon II X3 445 processor at 3.10 GHz, with 6 GB DDR3 RAM memory at 1333 MHz, a MSI 880GM-E41 motherboard, and a GeForce GTS 450 GPU with 1 GB of RAM.

The experimental analysis of the proposed algorithm executions was carried out on two platforms. Validation experiments using small-sized images were performed on the development platform, using a 500 GB SATA-2 disk with RAID 0 running Ubuntu Linux 11.10 64 bits Edition.

The limited computing power of the development platform did not allow taking full advantage of the parallel features proposed by the algorithm. Thus, a more comprehensive set of experiment including large images were performed in a more powerful platform, the *execution platform*, consisting in a Core i7-2600 processor at 3.40 GHz processor with 16 GB DDR3 RAM memory, with Fedora 15 64 bits Edition operating system, and a GeForce GTX 480 GPU with 1536 MB of RAM.

6.2 Problem Instances

Both sets of problem instances used for training and classification were obtained from the work by Shufelt and Mitchell [4]. These are images of different people in different poses. For each person and pose, the image comes in three sizes: 32×30 pixels, 64×60 and 128×120 pixels. There are about 620 images, which are divided in three sets, one for the network training and the other two to measure its effectiveness (trainlist and testlist, respectively). Also, a scaling of the images was performed in order to carry out executions with larger images to contrast these executions with the previous ones. The scaling was performed in two sizes: 256×240 and 512×480.

This variety of instance types (different image sizes) allows to take advantage of the GPU platform to the maximum because for instances of small size (i.e. 32×30 pixels, 64×60 pixels) both platforms perform similarly, while for images of larger sizes a clear difference between the two platforms begins to notice. This is mainly because the GPU platform has much more processing units than the CPU (yet at a lower speed), so if many calculations at once are required, there will be more available computing resources in GPU than in CPU.

6.3 Results and Discussion

To validate the algorithm it was considered that the rate of correctly classified images for new instances should be greater than 80% and the speedup gain over the sequential algorithm should be at least 2.

All values shown below correspond to algorithm executions with a training set consisting of 277 images (training set) and two test sets of 139 and 208 images respectively (train1 set and train2 set). In first instance, training over the ANN is made with every image in the training list, then performance is evaluated using images from test set 1 and 2 (this concludes a cycle). This is performed 100 times (100 epochs) to complete an execution cycle. The presented values correspond to the average of 50 execution cycles with an ANN with 100 neurons in the hidden layer.

Solution Quality

Since the proposed parallel implementation does not modify the algorithmic behavior of the sequential implementation, the obtained results with the GPU implementation are nearly the same than those obtained with the sequential version for all the studied instances. Table 2 and table 4 show correctly classified instances (in percentage) for both sequential and parallel algorithm and the learning rate and momentum constants that were used. Classification rates close to 80% are achieved for images of 512×480 pixels in both development and execution platform. For the development platform best results are obtained with images of 32×30 pixels where classification rate close to 93% is obtained as for the execution platform classification rate close to 92% is achieved for both 32×30 and 64×60 and 128×120 pixel images.

Execution Times

This subsection reports and discussed the execution time and performance results for the GPU implementation of the proposed algorithm. All execution times reported are the averages and its correspondent standard deviation values, computed in 50 independent execution of the parallel algorithm for each scenario.

Validation experiments on the development platform. Table 1 reports the execution times (in seconds) for the sequential and parallel implementations of the algorithm and the values of the speedup metric, in the development platform. Table 2 shows the classification rates obtained for both the sequential and parallel implementation, and the corresponding constants used for learning rate and momentum.

Table 1. Execution times (in seconds) in the development platform

image size	sequential	parallel	speedup
32×30	51.94 ± 0.57	56.22 ± 0.02	0.92
64×60	474.38 ± 18.90	147.54 ± 0.61	3.21
128×120	3665.99 ± 21.60	667.41 ± 2.26	5.49
256×240	16103.48 ± 80.41	3174.47 ± 1.84	5.07
512×480	67114.06 ± 132.66	14760.04 ± 1.55	4.54

Table 2. Correctly classified instances in the development platform

image size	sequential	parallel	learning rate	momentum
32×30	93.16%	92.91%	0.30	0.30
64×60	88.60%	88.83%	0.30	0.30
128×120	87.63%	86.79%	0.30	0.30
256×240	86.44%	86.56%	0.10	0.20
512×480	79.11%	79.80%	0.03	0.20

Experimental analysis on the execution platform. Table 3 reports the execution times for both sequential and parallel implementation, as well as the speedup obtained in experiments performed in the execution platform. Table 4 shows the classification rates obtained for each implementation and the corresponding constants used for learning rate and momentum.

Table 3. Execution times (in seconds) in the execution platform

image size	sequential	parallel	speedup
32×30	19.56 ± 0.24	34.29 ± 0.14	0.57
64×60	111.06 ± 2.94	61.69 ± 0.85	1.8
128×120	502.54 ± 7.95	224.09 ± 0.08	2.24
256×240	8633.87 ± 17.09	1019.13 ± 0.15	8.47
512×480	34540.95 ± 24.65	4777.90 ± 0.53	7.23

Table 4. Correctly classified instances in the execution platform

image size	sequential	sequential	learning rate	momentum
32×30	92.07%	91.96%	0.30	0.30
64×60	92.20%	91.84%	0.30	0.30
128×120	87.16%	91.35%	0.30	0.30
256×240	86.55%	86.55%	0.10	0.20
512×480	80.31%	78.63%	0.03	0.20

Speedup comparison. Fig. 7 summarizes the acceleration when using a GPU implementation, contrasted with using a sequential implementation in CPU, for the different image sizes and platforms used in the experimental analysis. The *speedup* evaluates the quotient between the execution time of the sequential implementation and the execution time in the parallel implementation in GPU.

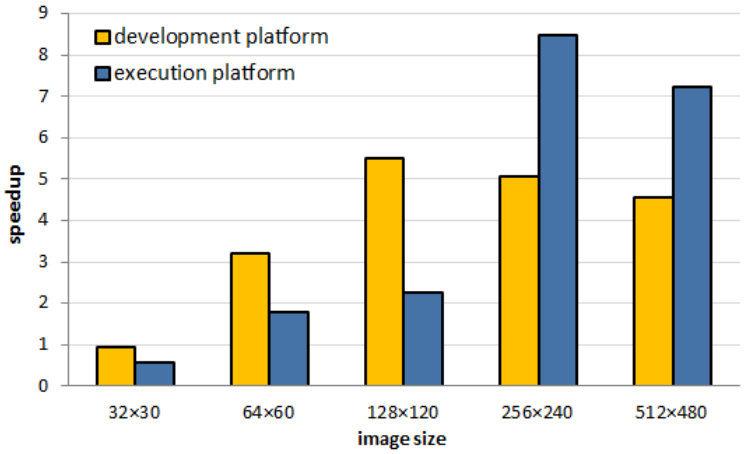

Fig. 7. Speedup comparison.

The speedup values in Fig. 7 indicate that the best acceleration is obtained for images with dimension 256×240 pixels, where the algorithm reaches the compute capabilities of the graphic card. The results in Tables 1 and 3 indicate that significant improvements on the execution times are obtained when using the parallel version of the algorithm with images of size 64×60 or larger. When solving images of size 32×30, the GPU implementation was unable to outperform the execution times of the CPU-only implementation, mainly due to the overhead introduced by thread creation and management and the use of the GPU memory. However, when solving larger problem instances, significant improvements in execution times are achieved, especially for images of size 256×240 pixels, where a speedup of 8.47 is obtained.

The previous results indicate that the parallel implementation of the face recognition algorithm executing on GPU provides significant reductions on the execution times over a traditional sequential implementation in CPU, especially when large images are processed.

7 Conclusions and Future Work

ANNs have proven to be suitable for solving many real world problems. However, the large execution times required in the training phase sometimes exclude ANNs from being an option when using large datasets or when solving complex problems.

Nowadays, parallel computing on GPUs allows achieving important performance improvements over CPU implementations. In this article, a parallel GPU algorithm was proposed for solving the face recognition problem using ANNs.

The parallel GPU algorithm was designed and implemented to take advantage of the specific features of GPU infrastructures, in order to provide an accurate and efficient solution to both the training process using the well-known backpropagation algorithm, and the face recognition problem itself.

The overall parallel strategy used is based on many threads running on GPU, each one working with several neurons, and trying to maintain the threads as independent as possible. Every kernel function was designed to take advantage of the execution platform, optimized to obtain the best performance (e.g. some kernels are assigned to perform more than one neuron calculations to avoid the overhead of thread creation). Also, shared memory was exploited in order to avoid global memory access latency.

The experimental analysis demonstrates that the parallel algorithm in GPU allowed obtaining significant improvements in the execution times when compared with a traditional sequential implementation. Speedup values up to **8.47** were obtained when solving problem instances with images of 256×240 pixels, and 7.23 for images of 512×480 pixels. These results confirm that to take advantage of the GPU computing power, the algorithm should be used to process images of considerable sizes.

The main contributions of this article include a parallel face recognition algorithm in GPU that is able to obtain accurate classification rates in reasonable execution times. The algorithm can be easily modified to recognize other features of a human face, without significant changes in the expected execution times.

The research reported in this article demonstrates that the GPGPU platform is a very good option to speed up the resolution of complex problems. Furthermore, the results indicate how the growing technological evolution of graphic cards helps to tackle more complex classification problems using ANNs, which can be solved accurately and in reduced execution times.

The main lines for future work include further improving the computational efficiency of the presented algorithm and tackling other classification/image processing problems using ANNs implemented on GPU. Regarding the first line of work, improved execution time results can be obtaining by adjusting the parameters of each kernel invocation, to avoid problems such as thread divergence or better use of the GPU resources (i.e. shared memory) for larger images. Also, some algorithm constants (such as *momentum* and *learning rate*) could be auto-tuned by the algorithm to obtain the best classification rates as possible. Regarding the second line, it will be of special interest to implement GPU algorithms to recognize generic features of people images, such as skin color, if it has sunglasses or not, etc., configurable at run time.

References

1. Kyong, K. and Jung, K.: GPU Implementation of Neural Network. Pattern Recognition, vol. 37, no. 6, pp. 1311-1314. Pergamon (2004)
2. Mitchell, Tom: Machine Learning. McGraw Hill (1997)
3. Bishop, Christopher M.: Pattern Recognition and Machine Learning. Springer (2006)
4. Mitchell, T. and Shufelt, J.: Neural Networks for Face Recognition, http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mitchell/ ftp/faces.html. Accessed June 2012.
5. Neural Network on GPU, http://www.codeproject.com/Articles/24361/A-Neural-Network-on-GPU Accessed June 2012
6. Jang, H., Park, A. and Jung, K.: Neural Network Implementation Using CUDA and OpenMP, Proc. of Computing: Techniques and Applications, pp.155-161. IEEE (2008)
7. Izotov, P., Kazanskiy, N., Golovashkin, D. and Sukhanov, S.: CUDA-enabled implementation of a neural network algorithm for handwritten digit recognition, Optical Memory & Neural Networks, vol. 20, no. 2, pp.98-106. Allerton Press, Inc (2011)
8. Nasse, F., Thurau, C. and Fink, G.: Face Detection Using GPU-Based Convolutional Neural Networks, Proc. of Computer Analysis of Images and Patterns, pp. 83-90. Springer Berlin (2009)
9. Lopes, N. and Ribeiro, B.: An Evaluation of Multiple Feed-Forward Networks on GPUs, International Journal of Neural Systems, vol. 21, no. 1, pp. 31-47. World Scientific Publishing Company (2011)
10. LeCun, Y., Bottou, L., Orr, G. and Muller, K.: Efficient Backprop in Neural Networks-Tricks of the Trade, Springer Lecture Notes in Computer Sciences, vol. 1524, pp. 5-50. Springer (1998)
11. NVIDIA. CUDA C Programming Guide Version 4.1, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf. Accesed June 2012
12. Steinkrau, D., Simard, P. and Buck, I.: Using GPUs for machine learning algorithms, Proc. of 8th Int. Conf. on Document Analysis and Recognition, pp. 1115–1119 (2005)
13. Catanzaro, B., Sundaram, N. and Keutzer, K.: Fast support vector machine training and classification on graphics processors, Proc. of 25th International Conference on Machine Learning, 2008, pp. 104–111. ACM (2008)
14. Rumelhart, D., Widrow, B. and Lehr, M.: The basic ideas in neural networks, Communications of the ACM, 37(3) pp. 87-92. ACM (1994)
15. Huang, S., Fu, L., Hsiao, P.: A framework for human pose estimation by integrating data-driven Markov chain Monte Carlo with multi-objective evolutionary algorithm, Proc. of Int. Conf. on Robotics and Automation, pp. 3748–3753 (2006)
16. Murphy-Chutorian, E., Trivedi, M.: Head Pose Estimation in Computer Vision: A Survey, IEEE Trans. on Patt. Analysis and Machine Intelligence, 2009, pp. 607–626. IEEE (2009)
17. Bishop, Christoper M.: Neural Networks for Pattern Recognition. Clarendon Press, Oxford (1995)
18. Yale Face Database, cvc.yale.edu/projects/yalefaces/yalefaces.html. Accessed June 2012
19. LeCun, Y. and Cortes, C.: The MNIST Database of Handwritten Digits, MNIST Handwritten Digit Database, http://yann.lecun.com/exdb/mnist. Accessed June 2012
20. CUDA Spotlights, http://developer.nvidia.com/cuda-spotlights. Accessed June 2012
21. Ng, C., Savvides, M. and Khosla, P.: Real-time face verification system on a cell-phone using advanced correlation filters, Proc. of 4th IEEE Workshop on Automatic Identification Advanced Technologies, pp. 57–62. IEEE (2005)
22. Venkataramani, K., Qidwai, S. and Vijayakumar, B.: Face authentication from cell phone camera images with illumination and temporal variations, IEEE Trans. on Systems, Man, and Cybernetics, Part C, vol. 35, pp. 411 – 418. IEEE (2005)