Comparative Study of Parallel Variants for a Particle Swarm Optimization Algorithm Implemented on a Multithreading GPU

Gerardo A. Laguna-Sánchez*[1], Mauricio Olguín-Carbajal[2], Nareli Cruz-Cortés[3], Ricardo Barrón-Fernández4, Jesús A. Álvarez-Cedillo[5]

[1,2,3,4]Centro de Investigación en Computación (CIC-IPN)
Instituto Politécnico Nacional, México D.F, Mexico
*galagunab07@sagitario.cic.ipn.mx

[5]CIDETEC-IPN
Instituto Politécnico Nacional, México D.F, México

ABSTRACT
The Particle Swarm Optimization (PSO) algorithm is a well known alternative for global optimization based on a bio-inspired heuristic. PSO has good performance, low computational complexity and few parameters. Heuristic techniques have been widely studied in the last twenty years and the scientific community is still interested in technological alternatives that accelerate these algorithms in order to apply them to bigger and more complex problems. This article presents an empirical study of some parallel variants for a PSO algorithm, implemented on a Graphic Process Unit (GPU) device with multi-thread support and using the most recent model of parallel programming for these cases. The main idea is to show that, with the help of a multithreading GPU, it is possible to significantly improve the PSO algorithm performance by means of a simple and almost straightforward parallel programming, getting the computing power of cluster in a conventional personal computer.

KEYWORDS. Multithreading GPU, PSO, general-purpose GPU, parallel programming, global optimization.

RESUMEN
El algoritmo Particle Swarm Optimization (PSO) ha tenido gran aceptación como alternativa de optimización global con base en heurísticas bio-inspiradas. Sus principales ventajas son su buen desempeño, baja complejidad computacional y un mínimo de parámetros. En general, las técnicas heurísticas han tenido un gran auge en los últimos veinte años y aún hoy resulta atractivo estudiar alternativas tecnológicas que permitan acelerar estos algoritmos para aplicarlos a problemas mucho más grandes y complejos. En este artículo se presenta un estudio empírico sobre la aplicación de algunas variantes paralelas para un algoritmo PSO, empleando un dispositivo de procesamiento gráfico (GPU) con capacidad multi-hilos y el más reciente modelo de programación paralela para estos casos. La idea principal es demostrar que es posible mejorar significativamente el desempeño del algoritmo PSO, mediante una programación paralela sencilla y directa, logrando con ello el poder computacional de un cluster en una computadora personal convencional.

Palabras clave: GPU con capacidad miltihilos, PSO, GPU para propósitos generales, programación paralela, optimización global.

1. Introduction

Some bio-inspired techniques, such as Evolutionary Computing [1], Ant Colony Optimization [2, 3], and PSO [4], were proposed as alternatives to solve difficult optimization problems obtaining acceptable solutions in a reasonable time. Due to these techniques work with a population of individuals, they simultaneously test different solutions based on specific rules and underlying stochastic processes. These heuristic techniques have been applied in practically all fields of knowledge, obtaining a good performance even running on common personal computers. The heuristic techniques obtain acceptable solutions in a “reasonably short time” compared with the traditional methods such as deterministic and enumerative techniques that may result impractical, specially while solving difficult optimization problems, since they explore each of the possible solutions of a problem. Nevertheless, the "reasonable" time that heuristic techniques may consume, this can be in the order of seconds, minutes, or even hours, depending on the problem to be solved. So, simpler algorithms, like PSO, have become very attractive because of their low computational complexity that leads to shorter execution times. However, when it is necessary to obtain a good real-time solution, even the simpler algorithms may seem to be slow. This situation has motivated the search for new ways to accelerate the performance of heuristic algorithms.

Recently, it was suggested to exploit the computational power available in the PC's graphic cards in order to solve general purpose problems [5] and the idea of general-purpose GPU (GPGPU) processing emerged. Since then, both manufacturers and developers have considered this new computing application as a promising research area, considering the wide range of possible applications that can take advantage of the parallelism available in the current GPUs.

Since bio-inspired algorithms were first reported, the idea of their parallelization was viewed as a natural consequence of their population-based feature [6]. In the case of the PSO algorithm, we can use parallel models developed for evolutionary algorithms, i.e.: global model, island model and diffusion model [7]. In this work we present some parallel variants for the PSO algorithm (two Global variants, and another one that we call embedded) implemented on a multithreading GPU. We report the obtained results using a new model of GPU programming that allows the programmer to write the code based on threads, additionally to the parallel operations on the graphic memory [8]. The main idea is to show that, with the help of a multithreading GPU, it is possible to improve the PSO algorithm performance, in a significant way, by means of a simple and almost straightforward parallel programming, getting the computing power of cluster in a conventional personal computer.

This work is organized as follows: In Section 2, we present the related work found in the specialized literature. In Section 3, we offer a brief description of the canonic algorithm for PSO. In Section 4, an outline of the general GPU architecture is given. In Section 5, we present practical considerations of our implementation. In Section 6 present experimental results. Finally, in Section 7, we draw our conclusions.

2. Related Work

Parallel programming usually involves migration of an existing sequential code towards concurrent, parallel or distributed architectures. The sequential PSO algorithm was not the exception and, after it was presented in 1995 [4], the first attempts to take advantage of its natural parallelism were reported like in the case of the work of J.F. Schutte in 2003 [9]. Interest in PSO parallelization is still a very current topic, which is shown by some recent research works that apply diverse parallel PSO algorithms to solve very complex optimization problems (i.g. see [10], [11], and [12]).

In the specialized literature, for a given algorithm, we can find proposals based on traditional concurrent processes, running in just one processor (e.g. see [18]), but most parallel implementations are usually designed to be executed in distributed systems (i.e. several processors in a network). In all these distributed systems, the communication overhead among different processors is a factor that considerably affects the performance of the parallel implementation. Because of that, it is understandable that some parallel implementations for PSO were proposed taking into account communication strategies, like in [13], [14] and [15]. Even new proposals for the PSO parallelization, based on the concept of parallel vectors, have been reported recently [16, 17].

Finally, we must note that although traditionally most of the early works related to the parallelization of a population-based algorithm with GPUs were firstly focused on the Evolutionary and Genetic Algorithms (e.g. see [19], [20], [21], and [22]), in some cases the resulting experiences were applied later to the PSO parallelization, like in [23].

Nowadays, specialized literature does not mention any empirical studies regarding the implementation of heuristic algorithms on a GPU with the new model of parallel programming based on multiple concurrent threads. This could be attributed to the fact that the programming tools were introduced very recently [24]. Therefore, this work represents the first empirical study comparing some parallel PSO variants on a multithreading GPU.

3. The Particle Swarm Optimization Algorithm

The PSO algorithm is based on the movement of particles or individuals that "fly" in an ***n***-dimensional space searching for a global optimum in a collaborative way. At each algorithm's iteration, the particles' position is updated following a simple rule where the individual's movement, although essentially random, is influenced by their own experience (individual learning) and by the environment (social influence) [25].

Originally, the algorithm was proposed by Kennedy and Heberhart [4], in 1995, based on the position x and change of position v (which was named velocity, by analogy) for every particle. Later, the algorithm was improved by Shi and Heberhart [26], in 1998, introducing the concept of inertia w. Denoting pbx the best fitness found so far by the particle, and gbx the best global fitness found so far within the population, then the PSO algorithm (in its canonical form) can be described as follows [27]:

Algorithm 1:

1.- Initialize every particle of the population in a random form, obtaining the values for the n-dimensional vectors of position x and velocity v.

2.- Calculate the fitness of every particle's position x_i. If the current fitness is better than pbx**,** then pbx is updated.

3.- Determine the location of the particle with the highest fitness and revise gbx.

4.- For every dimension d, of every particle i, the velocity v is updated according to the following equation:

$$v_{i,d}(t+1)=w\times v_{i,d}(t)+c_1\times r_1\times(pbx_{i,d}-x_{i,d}(t))+c_2\times r_2\times(gbx_{i,d}-x_{i,d}(t))$$

where w is the inertia of the system; c_1 and c_2 are constants that weigh the influence of individual learning and the social influence, respectively; and r_1 and r_2 are random variables, between 0 and 1, representing the free movement of every particle.

5.- Update the position x of each particle according to the following equation:

$$x_{i,d}(t+1)=x_{i,d}(t)+v_{i,d}(t+1)$$

6.- Repeat steps 2-5 until reaching the termination condition (number of iterations or precision).

4. GPU Architecture

The device selected to implement the parallel PSO variants is a GPU commonly used as graphic co-processor in systems with requirements of high performance video, such as video games. A GPU has its foundation on the vectorial processor architecture, which supports the execution of mathematical operations on multiple data in a simultaneous way. In contrast, the common CPU processors cannot handle more than one operation at the same time. Originally, the vectorial processors were commonly used in scientific computers [28], but later they were displaced by multi-nucleus architectures. Nevertheless, they were not completely eliminated because many computer graphics architectures, such as GPUs, use them as the basis for their hardware.

Usually, a conventional processor has to look for the next instruction to execute, which consumes time and generates latency during the instruction's execution. To reduce latency, modern processors execute a set of instructions in concurrent form, which is known as instruction pipelining, where the instructions go through several specialized subunits in turns: the first subunit reads the instruction and decodes it, the following subunit fetches it, and the last one executes the mathematical calculation. With instruction pipelining, decoding the next instruction starts before the first one has been carried out, in such a way that the instruction decoder is constantly used and the latencies are minimized. In these conditions, the execution time for a set of instructions is less than in a conventional processor, increasing the total throughput of the processor.

Vectorial processors lead the same idea further, pipelining both instructions and data. In this case, the same instruction operates on much data. This saves time while decoding instructions and results in a great computing power.

In computer graphics, the matricial representation is very common since images have a natural representation in this formalism and because operations on multiple sets of data are usually straightforward. This explains the wide acceptance of vectorial processors in the computer graphics area and their influence on the development of new hardware architectures destined to support the image generation required by video and gaming applications.

The big demand of gaming and video productions with real-time and photorealistic appearance has lead to the production of more powerful GPUs; prompting video-card manufacturers to develop multi-core based architectures and multi-thread proposals with computing power similar to the first Cray vectorial supercomputers.

Today, the trend in GPU development allows us to foresee the consolidation of a new model of parallel programming where the GPU does not only increase their original capacity of parallel calculation but rather has a more preponderant role as multithreading manager [29].

The GPU device used in this work was a GPU NVIDIA GeForce 8600GT with unified graphs and computing architecture that the manufacturer calls "Tesla", which is a scalable arrangement of multithreading multiprocessors. Every multiprocessor consists of eight processing cores, a multithreaded instruction unit and on-chip shared memory. A multiprocessor manages the creation, handling, and execution of the current threads in the hardware, supporting hundreds of threads (theoretically up to 512) under the SIMT approach (Single Instruction Multiple Treads) [29]. Since the multiprocessor maps every thread to a core and every thread is executed independently from the others, with its own instruction address and state registers, the NVIDIA programming tools offer some functions that are focused on the handling and optimization of multithreading.

5. Implementation Of Parallel Pso Variants On The GPU

Concerning parallelization models for the PSO algorithm, the same parallel classification suggested for the Evolutionary Algorithms was adopted as follows [7]:

1.- Parallel implementation with global approach. There is a main (master) processor and several slaves. In this model, the master distributes the work related to the fitness function evaluation to the slaves.

2.- Parallel implementation with migratory approach (also called *island approach*). In this model, the population is divided into several subpopulations (or *demes*) and the different processors run the same algorithm on each subpopulation. Eventually, after a certain time (called *epoch*), the processors commute to a stage of intercommunication where they interchange information and share the solutions found to that moment.

3.- Parallel implementation with diffusion approach. This may be seen as an extreme case of the island model where the population of every island is just one individual and where there are as many islands as existing individuals.

Based on the above classification, we programmed some parallel variants for the PSO algorithm as we will detail below. Our main objective was to program parallel implementations on the GPU using the CUDA (Computer Unified Device Architecture) software instead of complex and laborious methods based especially on parallel operations with graphic memory (i.e. textures, etc.). This was done in order to assess the performance gains that could be reached by the GPU through a relatively simple parallel programming strategy. Then, our implementations differ from the one presented in [23] because we do not use the traditionally GPU programming style based on parallel operations on multiple data, using an approach known as SIMD (Single Instruction Multiple Data) [8]. Instead, our implementations are based on the new model of parallel programming conceived to take advantage of the multithreading feature of NVIDIA GPUs, which allows us to manage multiple concurrent threads in a very efficient form. NVIDIA calls this feature "SIMT" (Single Instruction Multiple Threads) [29] and it is offered as a plus feature that complements the parallel operations of memory. This new programming model and its associated programming tool CUDA allows the programmers to write parallel programming on GPUs in a more natural way, turning GPUs in really general purpose programming tools. In fact, it is foreseen that this new model of parallel programming will be the reference for a future specification on a universal programming model that makes parallel programming possible, not only on GPUs, but on any multi-nucleus architecture or super computing platform that appears in the future [8, 24, 28].

In this work, three parallel variants for the PSO algorithm are reported: two Global variants and another one that we called embedded since it encloses both island and diffusion models. Concerning the global approach, two variants were programmed on the GPU:

1.- Global_ev: Where only the evaluation of objective function (fitness function) is parallelized.
2.- Global_ev+up: Where all the mathematical calculations are parallelized, computing all fitness function, velocity, position, and inertia.

Concerning the parallel implementation with the island model, and considering that the diffusion model is an extreme case of the island approach, we choose to include both variants under the same implantation and call it "*embedded* variant" because, in this case, the whole PSO algorithm runs on the GPU device and it appears as a black box from the host processor viewpoint.

The sequential PSO ***Algorithm 1*** was implemented as reference, in order to assess the performance of parallel variants. In all parallel implementations, the programming strategy involved the creation of one thread for each PSO particle. The rule was to replace all the sequential loops (specifically those where the iterations were in terms of the particles number) by a single multithreading kernel call. Thus, the sequential PSO algorithm and its parallel implementations have essentially the same structure. In the loops of the sequential code, each loop's iteration is independent from all others. Such loops can be automatically transformed into parallel kernels and each loop's iteration becomes an independent thread [8]. CUDA programs launch parallel kernels with the following extend function-call syntax:

kernel <<<dimGrid, dimBlock>>>(… parameter list …);

where **dimGrid** and **dimBlock** are specialized parameters that specify the dimensions of the parallel processing grid in blocks and the dimensions of the blocks in threads, respectively.

Figure 1 depicts the structure of the sequential PSO algorithm where the following functional blocks can be observed [25]:
- Population initialization. It initializes each particle of the population in a random form.
- Fitness function evaluation.
- Comparison. It determines if an individual has better fitness that the best registered.
- Imitation (updating). Every individual updates its position influenced by its own experience, and by the social environment.

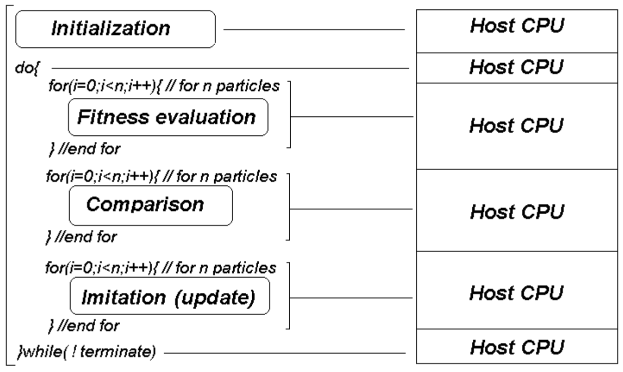

Figure 1. Structure of the sequential PSO algorithm

Intentionally, in order to illustrate the forward parallelization of the code, the sequential code has been organized highlighting the loops that are in terms of the particles number. The main idea is to create one thread for each PSO particle, as we will see below. Note that in the sequential PSO version (see Figure 1) all the functional modules are executed on the host processor. In the first parallel variant, the Global_ev one, only the fitness function evaluation module is parallelized (see Figure 2). Note that the respective loop was replaced by a kernel call that distributes the work to multiple threads on the GPU device. In the second parallel variant, the Global_ev+up one, any arithmetic calculation is distributed to the GPU, replacing both the fitness function evaluation and position update modules by the associated kernel calls (see Figure 3). Finally, in the third parallel variant, the embedded one, only the initialization module remains running on the host processor (see Figure 4), since there are kernel calls associated to the evaluation, comparison and imitation modules, that run on the GPU until a termination condition is reached.

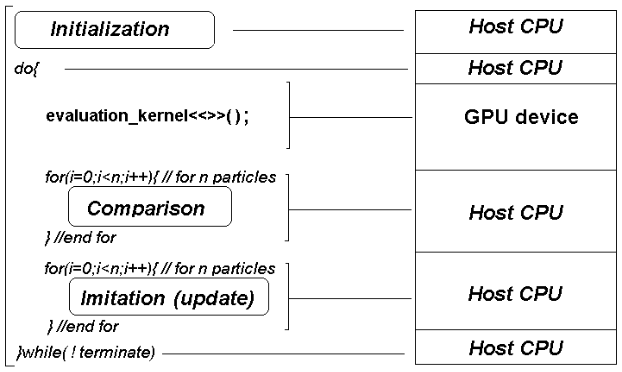

Figure 2. Structure of the Global_ev variant for the parallel PSO algorithm

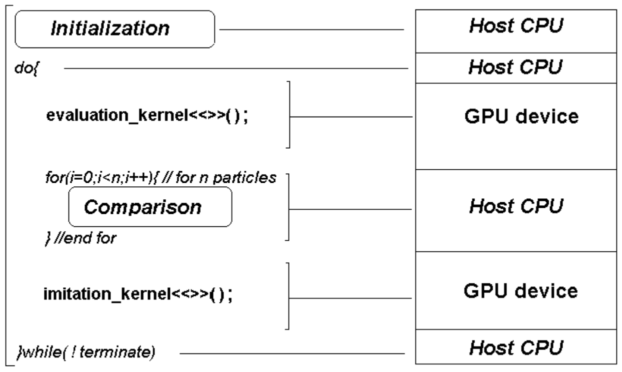

Figure 3. Structure of the Global_ev+up variant for the parallel PSO algorithm

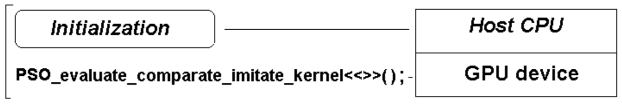

Figure 4. Structure of the embedded variant for the parallel PSO algorithm

Finally, there are some practical considerations that must be taken into account to achieve a functional implementation of the parallel PSO algorithm on a GPU:

Overhead. The GPU presents an overhead due to memory transferences between the host and the GPU device which are necessary during the information exchange. Because these transferences are relatively slow, any parallel implementation on a GPU must minimize their employment. Considering the overhead, it is understandable that the global variants (Global_ev and Global_ev+up) are slower than the embedded one, due to the information exchange between the host and the GPU during the algorithm execution. In the global variants, the information exchange in necessary since the host processor needs to know the information originated at both the evaluation and update modules in order to take any decision.

Synchronization. Before any decision branch, for example during the comparison process, all the running threads must be synchronized at the points where it is necessary to obtain unambiguous information. This point is particularly important when the threads have to communicate among themselves to share information. In these cases, it is a good practice to implement a master-slave communication strategy, additionally to the thread synchronization, in order to guarantee that only one of the threads is updating the global variables based on the information of all the others.

Contention. It is necessary to be careful when operating global variables that can be simultaneously revised by several threads. Appropriate precautions must be incorporated to deal with this problem. Specifically in the PSO algorithm, this situation happens with the variable that contains the index to the global best.

Random numbers generation. This is a fundamental topic for the correct operation of the PSO algorithm and it becomes critical when the random number generation is executed on the GPU. It has to be proved that any call to the *rand*() function running on the GPU will generate good random numbers (different numbers for every call and for every thread). If the above condition is not provided, it could result in a no converging algorithm because of the poor diversity. Thus, in the implementation of the embedded variant, the initialization module remains out of the GPU. Specifically, the seeds for random numbers (one per thread, in our case) are initialized on the host.

6. Experiments and Results

6.1 Experimental procedure

The experiments were conduced on a personal computer with a processor Intel Core Duo with a Linux operating system, which we name “the host processor”. The GPU is a graphic card NVIDIA GeForce 8600GT with 256 Mbytes of work memory and 4 multiprocessors, each one integrated by 8 cores, which represents a total of 32 processing cores. These processing cores were programmed by means of the CUDA environment which includes a new model of parallel programming that allows us to write the parallel code for the NVIDIA GPU in a straightforward way.

The objective was assessing the performance of our parallel PSO variants compared with the sequential one. The performance was measured according to the fitness function complexity, particles and iterations number. The implementations were tested solving some global numerical optimization functions from a well-known benchmark [30]. The following functions were selected because they are all multimodal and since they present significant complexity concerning fitness evaluation [29]:

F01 - Generalized Rosenbrock function.

$$f_1(x) = \sum_{i=1}^{n-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2 \right]$$

$$-30 < x_i < 30$$

$$\min(f_1) = f_1(1,...,1) = 0$$

with *n*=30,60,120 dimensions.

F02 - Generalized Rastrigin's function.

$$f_2(x) = \sum_{i=1}^{n} \left[x_i^2 - 10\cos(2\pi x_i) + 10 \right]$$

$$-5.12 < x_i < 5.12$$

$$\min(f_2) = f_2(0,...,0) = 0$$

with *n*=30,60,120 dimensions.

F03 - Generalized Griewank's function.

$$f_3(x) = \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

$$-600 < x_i < 600$$

$$\min(f_3) = f_3(0,...,0) = 0$$

with *n*=30,60,120 dimensions.

The PSO implementations tested in this work were
- Sequential variant. It is a sequential PSO algorithm that is executed completely on the host processor.
- Global_ev variant. In this variant, GPU only evaluates the fitness function and the remaining algorithm is executed on the host.
- Variant Global_ev+up. In this case, GPU evaluates the fitness function and executes the velocity/position update, while the remaining algorithm is executed on the host.
- Embedded variant. In this variant GPU executes most of the PSO algorithm. The host only executes the population initialization and results printing.

Then, the following experiments were carried out to measure the performance of each of the implemented PSO variants:
- Experiment 1. Comparative measurements of the processing time consumed by each of the functional modules (i.e. evaluate, compare, update) within the sequential PSO implementation.
- Experiment 2. Measurements of the processing time consumed by each of the PSO implementations, in terms of fitness function complexity.
- Experiment 3. For each of the parallel PSO variants, performance measurements in terms of iterations number.
- Experiment 4. For each of the parallel PSO variants, performance measurements in terms of particles number.

The objective of Experiment 1 was to measure, as a reference, the processing time consumed by each of the functional modules within the sequential PSO algorithm. This experiment was conducted in two sessions. In the first session, parameters were fixed to 30 dimensions, 20 neighbors, and 2000 iterations while the particles number was varied. In the second session, parameters were fixed to 256 particles, 20 neighbors, and 2000 iterations while the dimensions number was varied. The sequential PSO variant was executed 10 times for each of the 3 benchmark functions, obtaining the average consumed time, in seconds, for each of the functional modules and their respective proportion, as a percentage, in comparison with the total consumed time. The results of this experiment allowed us to empirically define the PSO modules that were more convenient to be parallelized in the case of global implantations.

The objective of Experiment 2 was to measure the processing time consumed by each of the PSO implementations (sequential and parallels) in terms of the fitness function complexity. Parameters were fixed to 10,000 iterations, 256 particles, and 30 dimensions. The PSO variants were executed 10 times for each of the F01, F02 and F03 benchmark functions, obtaining the average global optimum found and the average consumed time in seconds.

Experiments 3 and 4 were carried out to test the performance of the PSO variants concerning two fundamental parameters: iterations and particles number, respectively. The error and the consumed time in seconds are reported. During Experiment 3, while the iterations number was variable (10,000, 30,000, and 60,000 iterations), the following parameters were fixed:
Particles = 256
Dimensions = 60
Function = F03
Criterion of stop = iterations number.

Finally, during Experiment 4, while the particles number was variable (64, 256, and 1024 particles), the following parameters were fixed:
Iterations = 10,000
Dimensions = 60
Function = F03
Criterion of stop = iterations number.

Due to space constraints, and after observing that F03 takes more processing time than the sequential PSO variant, we decided to report only the results regarding the F03 optimization in order to illustrate the common behavior observed during the optimization of the three benchmark functions.

6.2 Performance metrics for parallel processing

In order to assess and plot the performance of all our parallel PSO variants, we defined the following metrics:
- Speedup measures the reached execution time improvement.
- Efficiency measures the use of the available processing cores.

Computational cost C is defined as the processing time (in seconds) that the PSO algorithm consumes. Then computational throughput V is defined as the inverse of the computational cost:

$$V = \frac{1}{C}$$

Speedup S is a rate that evaluates how rapid the variant of interest is in comparison with the variant of reference:

$$S = \frac{V_{obj}}{V_{ref}}$$

where V_{obj} is the throughput of the parallel implementation under study, and V_{ref} is the throughput of the reference implementation, i.e. the sequential implementation.

Finally, we define the parallel efficiency E as the rate resulting from dividing the speedup by the processing cores number:

$$E = \frac{S}{n}$$

where n is the processing cores number in the GPU (32 cores in our case).

6.3. Discussion of results

6.3.1. Experiment 1. Comparative measurements of the processing time consumed by each of the functional modules (i.e. evaluate, compare, update) within the sequential PSO implementation.

The experimental results showed that, after optimizing the three n-dimensional benchmark functions; the fitness function evaluation consumed a low proportion of the total processing time compared with the velocity and position update calculus (see Tables 1 and 2). This is due to the fact that the update operations, although with

Proportion of the total execution time with 30 dimensions, 2000 iterations						
Function	Particles					
	64		256		1024	
	Evaluation cost	Update-compare Cost	Evaluation cost	Update-compare cost	Evaluation cost	Update-compare Cost
F01	2.84%	96.59%	3.03%	97.25%	2.03%	97.28%
F02	27.54%	70.34%	26.93%	69.38%	27.16%	69.43%
F03	39.86%	55.24%	41.31%	55.81%	14.88%	35.93%

Table 1. Distribution of the computational cost as a function of the particles number

Proportion of the total execution time with 256 particles, 2000 iterations						
Function	Dimensions					
	30		60		120	
	Evaluation cost	Update-compare cost	Evaluation cost	Update-compare cost	Evaluation cost	Update-compare Cost
F01	3.03%	97.25%	3.26%	96.74%	3.38%	96.66%
F02	26.93%	69.38%	30.51%	67.81%	33.56%	66.53%
F03	41.31%	55.81%	44.92%	53.55%	46.95%	49.49%

Table 2. Distribution of the computational cost as a function of the dimensions Lumber

low arithmetical complexity, are computed more times than the fitness function evaluation. Furthermore, the process that generates the random numbers may consume a significant amount of processing time if the same processor executes the PSO algorithm and generates the random numbers. Thus, the experimental results allow us to conclude that for our three ***n***-dimensional optimization problems, most of the PSO processing time is consumed by updating (position and velocity) and not by the fitness function evaluating task.

6.3.2. Experiment 2. Measurements of the processing time consumed by each of the PSO implementations, in terms of fitness function complexity.

Concerning the time consumed, it was observed that the performance of the Global_ev variant may become worse than the sequential one if the arithmetical complexity of the objective function is small (see Figure 5). This amazing behavior (i.e. the Global_ev variant performance increases as the objective function complexity increases) is understandable if we think that a high arithmetic intensity (i.e. high number of arithmetic operations per memory operation) allows the GPU´s thread scheduler to overlap memory transactions (see [29], chapter 5). In comparison with the sequential variant, the experimental results show a better performance for the Global_ev variant in the case of the functions F02 and F03 optimization. Impressively, improvement is more notable for the Global_ev+up variant executing the F03 optimization (the most complex function) and is followed by results obtained executing the F01 and F02 optimizations (less complex functions). The best performance is achieved by the embedded variant executing the F03 optimization with an average execution time of 6 seconds, followed by the F01 and F02 cases. Again, note that the best performance is achieved executing the optimization of the most complex objective function. In comparison with the sequential variant, which consumes 35.9 seconds, the performance improvement of the embedded one is notable: only 6.6 seconds.

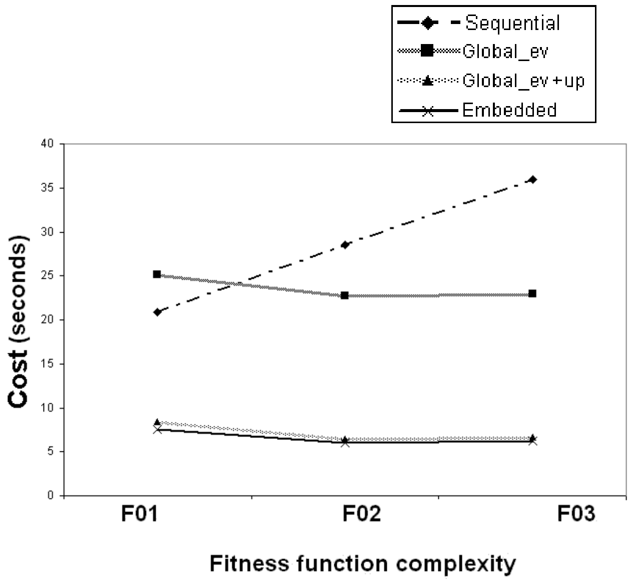

Figure 5. Performance of the PSO variants in terms of fitness function complexity. In this case, complexity(F01)< complexity(F02)< complexity(F03).

Thus, the experimental results allow us to conclude that a lower performance for the Global_ev and Global_ev+up parallel variants, in comparison with the embedded one, is due to the overhead produced by the information exchanges between the host computer and the GPU device.

6.3.3. Experiments 3 and 4. Performance measurements for each of the parallel PSO implementations in terms of iterations and particles number.

Concerning the error, we observed in all implementations that by increasing the iterations it was possible to obtain a global optimum solution very close to the real one. For example, setting 10,000 iterations as a starting point, it was observed that the sequential implementation has a low and very uniform error. It is notorious that the embedded variant begins with an error slightly greater than other implementations but, as the number of iterations increases, the solution reaches and improves the same precision for 30,000 and 60,000 iterations (see Figure 6), which is achieved in just a fraction of the time consumed by the sequential implementation with the same number of iterations. In fact, with 60,000 iterations the embedded variant has a lower error that the sequential one, with the same iterations, in a rate of 5 to 1 (see Figure 7).

In Figure 7, the consumed time as number of iterations increases is plotted. Note that improvement in the Global_ev+up variant is better than in the Global_ev variant and in the first one the GPU executes the velocity and position update in addition to the fitness function evaluation. Regarding the embedded variant, it is very interesting to note that the consumed time is practically the same, independently from the number of iterations.

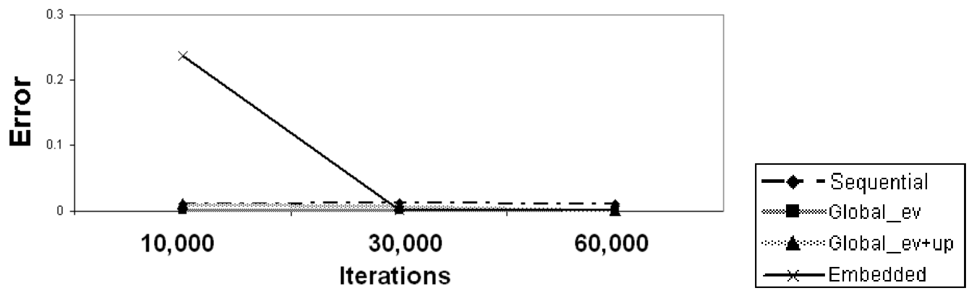

Figure 6. Error of the PSO variants, executing the F03 optimization with different iterations number

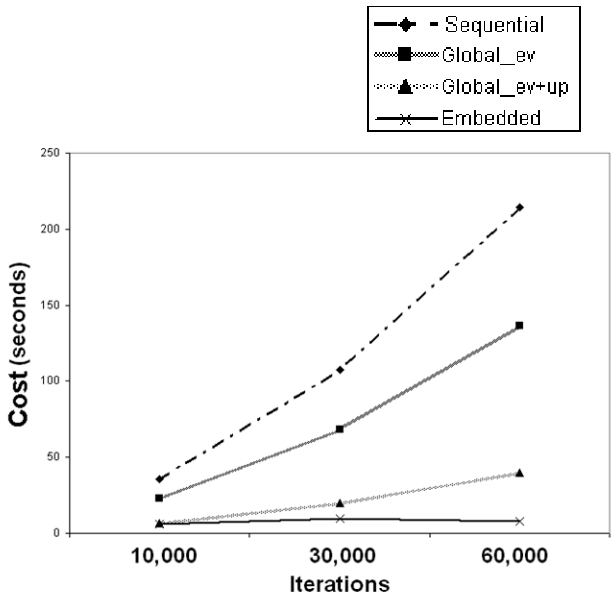

Figure 7. Consumed processing time by the PSO variants after executing the F03 optimization with different iterations number.

Experimental results show a similar behavior for the case where the particles number was varied (see Figure 8), i.e. performance of the parallel implementation is better on the GPU as the PSO particles number is increased. In general, theexperimental results showed that performance improves when the number of mathematical operations distributed to GPU is also increased.

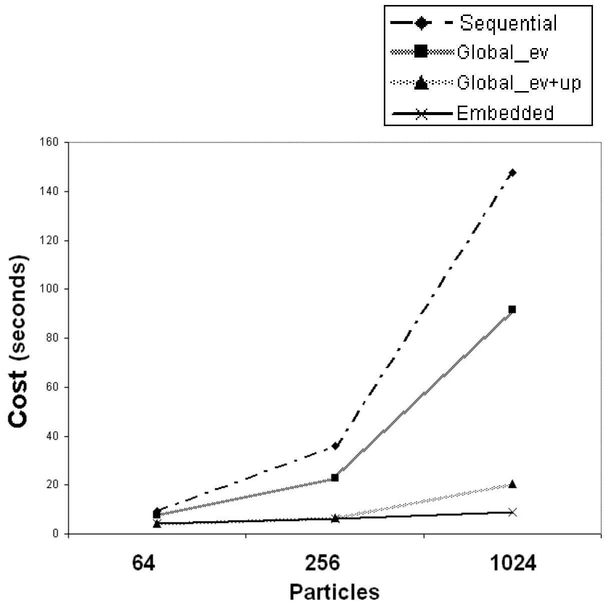

Figure 8. Consumed processing time by the PSO variants after executing the F03 optimization as a function of the particles number

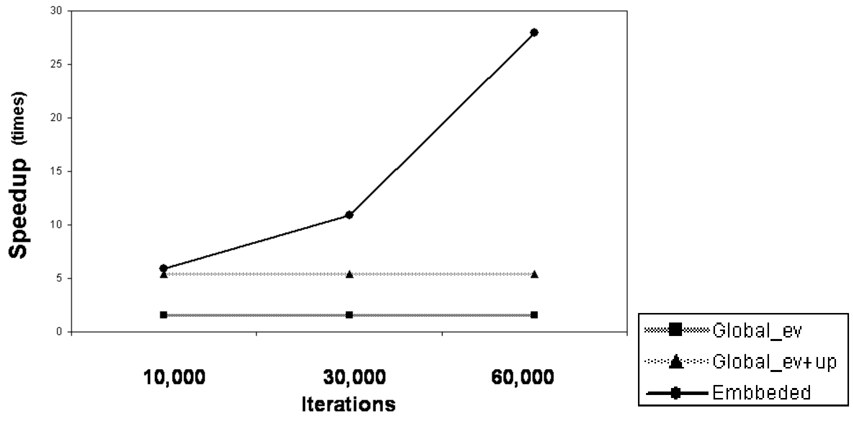

Figure 9. Speedup for the parallel PSO variants with different iterations number

6.3.4. Discussion on performance metrics

In general, the observed behavior is consistent for all tested functions. The experimental results allow us to conclude that the performance of the implemented algorithms acquires a notable improvement as more calculations are distributed to the GPU device. The performance metrics for all parallel PSO variants are depicted in Figures 9 to 12.

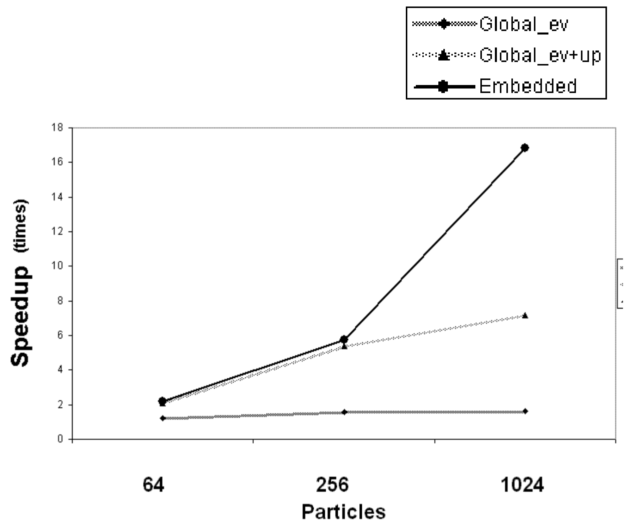

Figure 10. Speedup for the parallel PSO variants as a function of the particles number

Summing up, we can emphasize the better performance that GPU shows in relation to the host processor. For example, the optimization of the generalized Griewank's function (F03, of high computational cost for 60 dimensions, 60,000 iterations and 256 particles) takes the sequential variant 214.13 seconds, whereas it takes the Global_ev variant 136.3 seconds and the Global_ev+up one 39.89 seconds. Finally, the embedded variant was executed in only 7.66 seconds, which represents a speedup of 27.97 in comparison with the sequential one, resulting in an efficiency of practically 1 (remember that in our case the GPU is integrated by 32 cores).

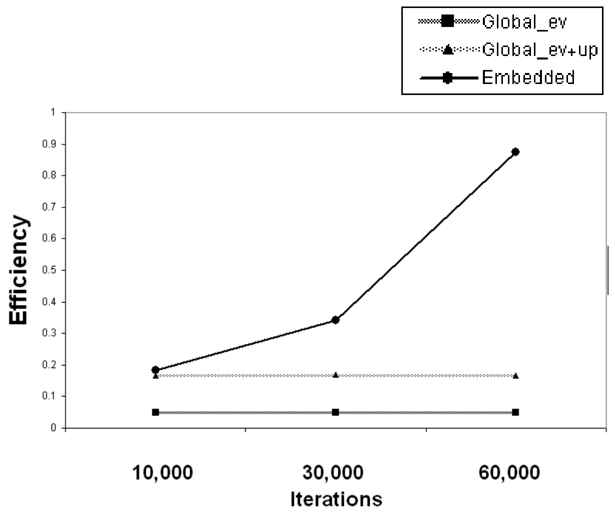

Figure 11. Efficiency for the parallel PSO variants with different iterations number

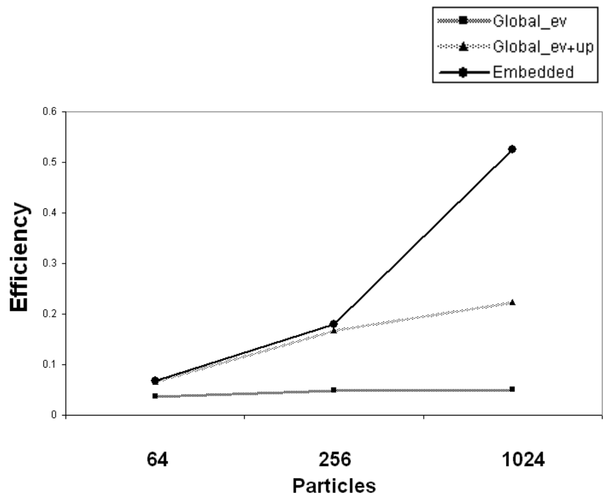

Figure 12. Efficiency for the parallel PSO variants as a function of the particles number

Note that by increasing the number of particles and iterations, the efficiency for the Global_ev and Global_ev+up variants keep practically constant, although the speedup metric increases. This is due to the fact that the overhead also increases proportionally to the number of particles and iterations. In contrast, the embedded variant increases its efficiency as it increases either the number of the particles or iterations.

7. Conclusion and Future Work

In this paper, an empirical and comparative study on the performance of three parallel variants for the Particles Swarm Optimization (PSO) algorithm implemented on a multithreading GPU, using CUDA as the most recent model of parallel programming, was presented. The experimental results showed that with these tools, it is possible to get some parallel variants for a given population-based algorithm with significantly improved performance by means of a simple and straightforward parallel programming.

In general, it was shown that a parallel programming based only on multithreading features of a GPU (i.e. exclusively distributing the work into threads and without using parallel operations on multiple data) at least results in a speedup proportional to the number of GPU cores, getting the computing power of cluster in a conventional personal computer. It is supposed that using the whole GPU capacities, executing parallel operations on multiple data in addition to the multithreading feature, it is possible to increase the performance even more. Also, it was shown that the whole performance of GPU improves when the quantity of simple tasks distributed to GPU threads is increased. This behavior can be appreciated in the performance of all three parallel variants tested, but it is more evident in the embedded one, where its performance was significantly improved.

Future research may be focused on the following three points: (1) To apply the whole multithreading GPU capacities, including the parallel operations on multiple data in addition to the multithreading feature, to get a powerful high parallelized PSO algorithm. (2) The necessity to test more fitness functions. (3) To use GPU and the new parallel programming model CUDA to deal with the parallelization of other population-based algorithms, e.g. the Differential Evolution algorithm.

Finally, the authors foresee that in the future the development of GPUs and their new parallel programming models will lead to the availability of scientific super-computing on conventional PCs, thanks to the new hardware platforms inspired on the modern GPUs and its predecessors: the vectorial processors.

References

[1] A. E. Eiben and J.E. Smith, Introduction to Evolutionary Computing, Natural Computing Series Springer, 2003.

[2] M. Dorigo and T. Stützle, The Ant Colony Optimization Metaheuristic: Algorithms, Applications, and Advances, Handbook of Metaheuristics, Kluwer Academic Publishers, 2002, pp. 251—285.

[3] M. Dorigo and K. Socha, Ant Colony Optimization, in Handbook of Approximation Algorithms and Metaheuristics, T.F. Gonzalez, Ed., Chapman & Hall, 2007, pp. 26.1—26.14.

[4] J. Kennedy and R. Eberhart, Particle Swarm Optimization, in Proceedings of IEEE Conference on Neural Networks, 1995, pp. 1942–1948.

[5] J. Owens et al., A Survey of General-purpose Computation on Graphics Hardware, Computer Graphics Forum, Vol. 26, No. 1, 2007, pp. 80–113.

[6] E. Cantú-Paz, Efficient and Accurate Parallel Genetic Algorithms, Kluwer, 2000.

[7] M. Belal and P. El-Ghazawi, Parallel Models for Particle Swarm Optimizers, International Journal of Intelligent Computing and Information Sciences, IJICIS, Vol. 4, No. 1, January, 2004, pp. 100–111.

[8] J. Nickolls et al., Scalable Parallel Programming with CUDA, ACM Queue, 2008, pp. 40–53, March/April, on line at: http://mags.acm.org/queue/20080304/?u1=texterity.

[9] J.F. Schutte et al., Parallel Global Optimization with the Particle Swarm Algorithm, International Journal for Numerical Methods in Engineering, Vol. 61, No. 13, 2003, pp. 1–24.

[10] N. Jin and Y. Rahmat-Samii, Parallel Particle Swarm Optimization and Finite-difference Time-domain (PSO/FDTD) Algorithm for Multiband and Wide-band Patch Antenna Designs, IEEE Transactions on Antennas and Propagation, Vol. 53, No. 11, November, 2005, pp. 3459–3468.

[11] S. Cui and D. Weile, Application of a Parallel Particle Swarm Optimization Scheme to the Design of Electromagnetic Absorbers, IEEE Transactions on Antennas and Propagation, Vol. 53, No. 11, November, 2005, pp. 3616–3624.

[12] H. Ma and et al., Research on Parallel Particle Swarm Optimization Algorithm Based on Cultural Evolution for the Multi-level Capacitated Lot-sizing Problem, in Proceedings IEEE Control and Decision Conference, 2008, pp. 965–970, July.

[13] J. Chang et al., A Parallel Particle Swarm Optimization Algorithm with Communication Strategies, Journal of Information Science and Engineering, Vol. 21, No. 4, 2005, pp. 809–818.

[14] L. Bo et al., Parallelizing Particle Swarm Optimization, IEEE Pacific Rim Conference on Communications, in Computers and Signal Processing, PACRIM, 2005, pp. 288–291, Aug.

[15] S. Mostaghim et al., Multi-objective Particle Swarm Optimization on Computer Grids, in Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, 2007, pp. 869 – 875, July.

[16] K. Parsopoulos et al., Multiobjective Optimization Using Parallel Vector Evaluated Particle Swarm Optimization, in Proceedings of the IASTED International Conference on Artificial Intelligence and Applications (AIA), 2007, pp. 869 – 875, July.

[17] I. Schoeman and A. Engelbrecht, A Parallel Vector-based Particle Swarm Optimizer, Adaptive and Natural Computing Algorithms, in Proceedings of the International Conference in Coimbra. 2005, pp. 268–271.

[18] S. Baskar and P Suganthan, A Novel Concurrent Particle Swarm Optimization, in Proceedings IEEE Congress on Evolutionary Computation, 2004, pp. 792–796.

[19] S. Harding and W Banzhaf, Fast Genetic Programming on GPUs, in Proceedings of the 10th European Conference on Genetic Programming, ser. Lecture Notes in Computer Science, M. Ebner et al., Eds., Vol. 4445, 2007, pp. 90–101, April.

[20] S. Harding and W. Banzhaf, Fast Genetic Programming and Artificial Developmental Systems on GPUs, in Proceedings of the 21st International Symposium on High Performance Computing Systems and Applications, 2007, pp. 2–2, May.

[21] W. B. Langdon and W. Banzhaf, A SIMD Interpreter for Genetic Programming on GPU Graphics Cards, in EuroGP, ser. LNCS, Vol. 4971, March, 2008, pp. 73–85.

[22] D. Robilliard et al., Population Parallel GP on the G80 GPU, in Lecture Notes in Computer Science, Vol. 4971, 2008, pp. 98–109.

[23] J. Li et al., An Efficient Fine-grained Parallel Particle Swarm Optimization Method Based on GPU-acceleration, International Journal of Innovative Computing, Information and Control, Vol. 3, No. 6(B) , Dec., 2007, pp. 1707.

[24] T. Halfhill, Parallel Processing with CUDA, Microprocessor Report, www.MPRonline.com, 2008. On line at: http://www.nvidia.com/docs/IO/55972/220401_Reprint.pdf, Consulted on Feb. 19th 2009.

[25] J. Kennedy and R. Eberhart, Swarm Intelligence, Morgan Kaufmann Academic Press, 2001.

[26] Y. Shi and R. Eberhart, A Modified Particle Swarm Optimizer, in IEEE International Conference on Evolutionary Computation Proceedings, 1998, pp. 69–73, May.

[27] M. O'Neill, and A. Brabazon, Self-Organising Swarm (SOSwarm), Soft Comput, Vol.12, January, 2008, p.p. 1073-1080.

[28] A. Mzoughi, O. Lafontaine, and D. Litaize, Performance of the Vectorial Processor VECSM2* Using

Serial Multiport Memory, in Proceedings of the 10th International Conference on Supercomputing, 1996, pp. 390–397, Toulouse Cedex, France.

[29] CUDA: Computer Unified Device Architecture Programming Guide, Version 2.0, NVIDIA Corporation, California, USA, 2008, July.

[30] E. Mezura-Montes, J. Velázquez-Reyes and C. A. Coello-Coello, A Comparative Study of Differential Evolution Variants for Global Optimization, GECCO, 2006, pp. 485–492, July.

Acknowledgments
G.A. Laguna-Sánchez and R. Barrón-Fernández acknowledge Instituto Politécnico Nacional (IPN) and Consejo Nacional de Ciencia y Tecnologíal (CONACyT) of Mexico for their support to this research through SIP-20090620 project (IPN), I0013/91434 fund (CONACyT), and Ph. D. scholarship # 210397 (CONACyT).

Authors´ Biography

Gerardo Abel LAGUNA-SÁNCHEZ

He received the M.S. degree (1998) from Universidad Nacional Autónoma de México (DEPFI-UNAM); the B.Sc. degree in electronic/computer engineering (1993) from Universidad Autónoma Metropolitana (UAM), Mexico City; and the Electronic Technician degree(1988) from "Wilfrido Massieu" CECyT-IPN, Mexico City. Currently, he is pursuing the PhD degree and is with the Artificial Intelligence (AI) Laboratory at the Centro de Investigación en Computación (CIC) at Instituto Politécnico Nacional (IPN), Mexico. His research interest includes AI and advanced signal processing applications. From 1993 to 2007, he worked at the Mexican electrical industry as a D&D engineer designing and developing SCADA systems and OEM devices.

Mauricio OLGUÍN-CARBAJAL

He received the M.Sc (2000) degree in computer engineering from CIC, IPN, Mexico and the B.Sc. (1995) degree in electrical and electronic engineering from ESIME Culhuacán, IPN, Mexico. Currently, he is a professor and researcher at the CIDETEC-IPN, in Mexico City, Mexico, which he joined in 2002. Nowadays, he is pursuing his doctoral studies in computer graphics at CIC. His mayor research interests are in computer graphics, virtual reality, evolutionary computation, tetworking and network Security.

Nareli CRUZ-CORTÉS

She received the PhD degree in electrical and computer engineering (2004) from CINVESTAV, Mexico; the M.Sc. degree in artificial intelligence (2000) from the Universidad Veracruzana and the LANIA, Veracruz, Mexico; and the B.Sc. degree in computer engineering (1995) from the Instituto Tecnológico de Tepic, Nayarit, Mexico. Currently, she is with the Artificial Intelligence (AI) Laboratory at CIC-IPN where she is a researcher and professor. Her major interests are in combinatorial and multiobjective optimization, evolutionary algorithms and bio-inspired algorithms.

Ricardo BARRÓN-FERNÁNDEZ

He received the M.S. (1998) and PhD (2007) degrees in computer science from Centro de Investigación en Computación (CIC) at Instituto Politécnico Nacional, and the B.Sc. degree in mathematics (1985) from Universidad Nacional Autónooma de México (UNAM). Currently, he is with Artificial Intelligence (AI) Laboratory at CIC where he is a researcher and professor. His research interests includes signal processing, geometric algebra and AI applications.

Jesús Antonio ÁLVAREZ-CEDILLO

He is an ICE engineer who graduated from Instituto Politécnico Nacional (1987). He received the M.S. degree in computer science (2002) from CINTEC-IPN, Mexico. Currently, he is pursuing the PhD degree and he is with CIDETEC-IPN. His areas of interests are in computer architectures, parallel and distributed architectures, operating systems for parallel and distributed systems, graphing-modeling, and town memory GPUs.