Journal of Theoretical and Applied Information Technology
20th April 2016. Vol.86. No.2
© 2005 - 2016 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

A DIFFERENTIAL EVOLUTION ALGORITHM PARALLEL IMPLEMENTATION IN A GPU

[1]LAGUNA-SÁNCHEZ G. A., [2]OLGUÍN-CARBAJAL M., [3]CRUZ-CORTÉS N., [4]BARRÓN-FERNÁNDEZ R. AND [5]CADENA MARTÍNEZ R.
[1]Universidad Autónoma Metropolitana, UAM
[2,3,4]Instituto Politécnico Nacional, Department of postgraduated, IPN
[5]Universidad Tecnológica de México, UNITEC
E-mail: [1]glaguna@xanum.uam.mx, [2]molguinc@ipn.mx, [4]rbarron@cic.ipn.mx, [5]rocadmar@mail.unitec.mx

ABSTRACT

The computational power of a Graphics Processing Unit (GPU), relative to a single CPU, presents a promising alternative to write parallel codes in an efficient and economical way. Differential Evolution (DE) algorithm is a global optimization based on bio-inspired heuristic. DE has a good performance, low computational complexity and need few parameters. This article presents parallel implementation of this population-based heuristic, implemented on a NVIDIA GPU device with multi-thread support and using CUDA as the model of parallel programming for these case. Our goal is to give some insights about GPU's parallel programming by a simple and almost straightforward parallel code, and compare the performance of DE algorithm running on a multithreading GPU. This work shows that with a parallel code and a NVIDIA GPU not only the execution time is reduced but also the convergence behavior to the global optimum may be changed in a significant manner with respect the original sequential code.

Keywords: *Multithreading, Parallel Programming, GPU, Differential Evolution And Fine Grain.*

1. INTRODUCTION

Today, the idea of exploiting the computational power available in the PC's graphic cards in order to solve general purpose problems [1] and the general-purpose GPU (GPGPU) processing concept are current topics. 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 low price GPUs.

Since parallelization of some bio-inspired algorithms is viewed as a natural consequence of their population-based feature, recently it was shown in [2] that it is possible to reach a significant speedup for Particle Swarm Optimization (PSO) algorithm when it is parallelized and executed on a multithreading GPU, after a simple and almost straightforward parallel programming style supported by the CUDA programming tool [3]. In [2] the authors showed that the best performance was reached when the execution of the whole PSO algorithm was delegated to the GPU following an approach similar to that known as diffusion within the parallel programming community [4]. In that work the authors called their parallel implementation embedded because in the diffusion approach there is one processor per individual but in the proposed model there is one thread instead of one processor per individual.

Bio-inspired techniques such as Evolutionary Computing [5], Ant Colony Optimization [6] and Differential Evolution (DE) [3] were proposed as alternatives to solve difficult optimization problems obtaining acceptable solutions in a reasonable time. Since 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.

In this paper a parallel version DE algorithm for a multi-threading GPU is presented and the performance of such parallelized version are compared and reported as a continuation of the earlier research work in [2]. The DE algorithm was chosen since it is very popular for optimization purposes and new versions are emerging continuously as microDE [7] [8], Adaptive DE [9] and DE with Thresheld Convergence [10] among many others. The computational power provided by the GPU results in a natural speedup but shows that, additionally, the proposed parallel implantation have different behavior, compared to sequential one, as a result of the specific way in which the random numbers are generated within GPU.

This work is organized as follows. Section II presents a brief overview of related work. Section III presents an introduction to GPU architecture. Section IV offers a brief description of the DE algorithm. Section V presents practical considerations of our parallel implementation. Section VI reports experimental results. Finally, in Section VII we draw our conclusions.

2. RELATED WORK

Parallel programming usually involves migration of an existing sequential code towards concurrent, parallel or distributed architectures. Concerning population-based algorithms (like Genetic algorithms, PSO, DE, etc.), once they were presented, there were attempts to take advantage of its natural parallelism, or example, the works of Cantú-Paz [4] and J.F. Schutte [11]. In specialized literature we can find proposals based on traditional concurrent processes, running in just one processor, 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 consider ably affects the performance of the parallel implementation. So far, interest in parallelization of population-based algorithms is still topical, which is shown by some recent research works that propose diverse parallel implementations of these kind of bio-inspired heuristics in order to solve very complex optimization problems (see [12] and [5]).

Regarding parallelization of population-based algorithms on GPUs, the first proposals were focused on Genetic Programming (see [13]) and in some cases the resulting experiences were applied later to parallelization of other population-based algorithms, like in [14]. Recently, in [2] the authors proposed exploiting the advantages of a NVIDIA multithreading GPU and CUDA programming tool for parallelization of a PSO algorithm in a simple and straightforward way. This work presents a first empirical study comparing sequential DE algorithm against they parallel variant running on a multithreading GPU.

3. INTRODUCTION TO GPUS AND MULTITHREADING ARCHITECTURE

The modern GPUs have their foundation on the vectorial processor architecture, which supports the execution of mathematical operations on multiple data in a simultaneous way. In contrast, the original CPU processors cannot handle more than one operation at the same time. Originally, the vectorial processors were commonly used in scientific computers [15], but later they were displaced by multi-nucleus architectures.

Nevertheless, the vectorial processors were not completely eliminated, since many computer graphics architectures and the modern GPUs are essentially inspired by them.

3.1 CUDA architecture

CUDA programming tool is modeled by a single instruction multiple thread (SIMT) approach where multiple threads are executed on many data elements. CUDA allows programmers to write parallel code using standard C language with NVIDIA extensions. CUDA organizes parallelism in a hierarchal system of three levels: grid, block, and thread. The process begins when the host (CPU) invokes a GPU device function called kernel, then a grid of multiple thread blocks is created in order to be distributed to available multiprocessors. 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.

During kernel execution, threads have access to five types of GPU memories, depending on a defined hierarchy or access levels (see Fig. 1):

• Global memory, a read/write memory located on GPU board.

• Constant memory, a read cached memory located on GPU board.

• Local memory, a per-thread read/write memory located on GPU board.

• Shared memory, a per-block read/write memory located on GPU chip.

• Register memory, the fastest per-thread read/write memory located on GPU chip.

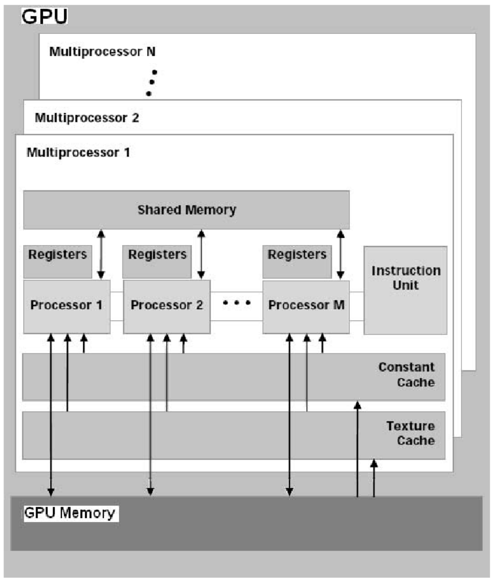


Fig. 1. Memory Hierarchy Of NVIDIA GPU [3]

Shared memory and registers are the fastest but limited in size because they are on-chip memory. On the other hand, device memory (Local, Global and Constant are on-board memory) are large but are accessed with high latency compared with on-chip memory. Since the multiprocessor executes threads in 32 parallel thread groups, called warps, the global memory can be efficiently accessed by threads in half-warp by simultaneous memory read/write coalesced into a single memory transaction of 32, 64, or 128 bytes [3].

4. OVERVIEW OF DIFFERENTIAL EVOLUTION ALGORITHM

DE algorithm is a population-based algorithm that have attracted the attention of many researchers. Although DE general structure is very similar to other population based algorithms(initialization, fitness evaluation, comparison and updating blocks), DE uses a unique rules for offspring generation, comparison and updating. In a general sense in DE strategy the individuals form teams of just three individuals in order to create a new individual (using a kind of recombination and mutations operators) that tries to improve the current population's best individual. From the No Free Lunch Theorem [16] we know that using a limited set of benchmark functions does not guarantee that an algorithm that performs well on them will be well comported in a different set of problems. In fact it is known that each heuristic is competitive in specific kinds of problems. So our goal is to give some insights about DE algorithm exploit potential regards the GPU to potentiality reduce the convergence time, based on the type of problem and the dependence of execution time as a function of the individuals or iterations number.

4.1. Differential Evolution Algorithm (DE)

The main idea emerged when Price and Storm [17] proposed to use vector differences to disturb population vectors to fit parameters for Chevichev polynomials. Differential Evolution (DE) is a bio-inspired optimization heuristic and population-based algorithm that uses mutation, crossover, and selection operators, to evolve individuals. DE basic idea relies on generation of test and trial vectors. In DE is found a vector:

$$x_{i,G} = 0, 1, 2, ..., Np \qquad (1)$$

as a tentative solution to the problem. Where **Np** does not change during the algorithm execution. The population is constructed with all *xi,G* vectors from 1 to *n*:

$$PG = \{x_{(1,G)}, ...x_{(n,G)}\} \qquad n \,\square\, [1, Np] \qquad (2)$$

Storn and Price highlighted variants for the DE algorithm.

The different schemes for naming DE variants, *DE/x/y/z* where:

1) **DE** denotes a Differential Evolution Algorithm.

2) **x** is the mechanism to select a vector X_{r1} .

3) **y** is the number of weighted difference vectors F (X_{r2}– X_{r3}) used to perturb X_{r1}.

4) **z** is the crossover scheme.

For this work it uses DE/rand/1/bin, this refers to a Differential Evolution with a random selected vector (rand) using one weighted difference vector and a binomial crossover scheme. The crossover operator works by mixing components of the current and mutated elements to construct an offspring.

DE has two main crossover variants, binomial and exponential. DE can have different mutation probabilities depending of the crossover variant implemented; this will be reflected on the DE behavior [18].

DE population evolves by using three operators: mutation, crossover and selection.

The **mutation** operator used was as follows:

$$v_G = x_{r0,G} + F \times (x_{r1,G} - x_{r2,G}), \qquad (3)$$

with r_0, r_1, r_2 □ $[1, Np]$, $r_0 = r_1 = r_2$

were r_0, r_1, r_2 are random selected individuals, and r_2 is used for the base vector $x_{r2,G}$. F is a used as a factor that controls the difference between vectors. This means that for each $x_{i,G}$ in the population, a noisy vector ***v*** is generated.

The crossover operator is applied on the noisy vector vG and the population vectors, obtaining a trial vector ui,G , for i □ [1, Np]. The vector

$$u = (u_0 , u_1, \ldots, u_{D-1}) \qquad (4)$$

$$u_{ij,G+1} = \begin{cases} v_{ij,G+1}, & \text{if} \quad R_j \le CR \quad or \quad j = r_k \\ x_{ij,G}, & \text{if} \quad R_j > CR \quad and \quad j \ne r_k \end{cases} \qquad (5)$$

with

where CR □ [0, 1] is a crossover constant for the new vector $u_{i,G}$ generation; j = 1, 2, ..., n; Rj is a random number of a uniform random number generator in [0,1]. Samples were renewed for every component of the trial vector **v**, while r_k □ [1, Np] is the random uniform individual index, and $x_{i,G}$ □ [1, Np] is a population selected individual.

In the selection step, the trial vector $u_{i,G}$ is compared with the actual $x_{i,G}$ selected parent. The one with the best fitness passes to the next generation, see Eq. 6:

$$\vec{x_{i,G+1}} = \begin{cases} \vec{u_{i,G}}, & \text{if} \quad f(\vec{u_{i,G}}) \le f(\vec{x_{i,G}}) \\ \vec{x_{i,G}}, & \text{if} \quad f(\vec{u_{i,G}}) > f(\vec{x_{i,G}}) \end{cases} \qquad (6)$$

From Eq. 3 can be seen that, in order to allow the application of the mutation operator, a DE algorithm must have at least 4 individuals. Finally, it is important to remark that for different kinds of problems, in order to obtain better results, it is convenient to have specific and fixed algorithm's parameters [19].

5. PARALLEL IMPLEMENTATION WITH CUDA

Concerning parallelization models for population-based algorithms, the parallel classification suggested for Evolutionary Algorithms in [20] as global approach, migratory approach,and diffusion approach was firstly considered as starting point reference at the beginning of the research [2]. In that work was shown that a convenient and straightforward way for parallelization of population-based algorithms, in a multi-threading GPU, is a kind of diffusion implementation where there is a GPU thread by each individual, in such a way that single individual's fitness evaluation and individual's update are executed by a single GPU thread while the comparisons among the individuals are carried out within GPU after a convenient threads synchronization. Such implementation is called Embedded by the authors, since most of functional blocks, except initialization one, are delegated to GPU. Thus, embedded approach was chosen as programming model for the parallel implementation of DE algorithm on the NVIDIA GeForce 8600GT GPU.

DE heuristic described in Algorithm 1 were parallelized by mean of a programming strategy that consists in the creation of one thread for each individual and a kernel call in order to execute the code's body of a given population-based

Algorithm 1 Pseudocode for Differential Evolution

```
1:  $P_0 \leftarrow GenerateRandomlyInitialPopulation \quad i = 1...Np$
2:  $Max_G \leftarrow Number\ of\ generations$
3:  $CR \leftarrow 0.9$
4:  for $G = 1$ to $Max_G$ do
5:      $P_G \leftarrow$GenerateWorkPopulation($P_{G-1}$);
6:      EvaluateWorkPopulation($P_G$);
7:      for $n = 1$ to $Np$ do
8:          $Randomly\ select \quad i \ne r1 \ne r2 \ne r3$
9:          $\vec{u_{i,G}} \leftarrow$ CalculateTrialVector($\vec{x_{i,G}}, \vec{x_{r_1}}, \vec{x_{r_2}}, \vec{x_{r_3}}$);
10:         $\vec{x_{i,G+1}} \leftarrow$ TestTrialVector($\vec{x_{i,G}}, \vec{u_{i,G}}$);
11:         $I_G \leftarrow I_G + 1$
12:     end for
13:     $S_{ellite} \leftarrow$SelectEllitist($P_G$);
14:     $E_G \leftarrow E_G + 1$
15: end for
```

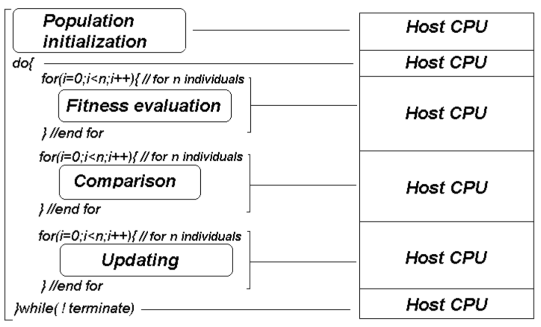


Fig. 2. Structure Of DE Sequential Algorithm

algorithm. Thus, while in the sequential code each particle movement is updated particle by particle, within GPU all particles updating is executed in a concurrent way. CUDA programming tool allows programmer to launch a kernel and creating a block of independent threads that represent the population and individuals, respectively. Specifically, a CUDA program includes the following steps:

1) Allocate memory on the GPU device.

2) Copy data from host (CPU) into GPU memory.

3) Host invokes kernel function.

4) GPU executes the code.

5) Copy the output results back from GPU memory into host memory.

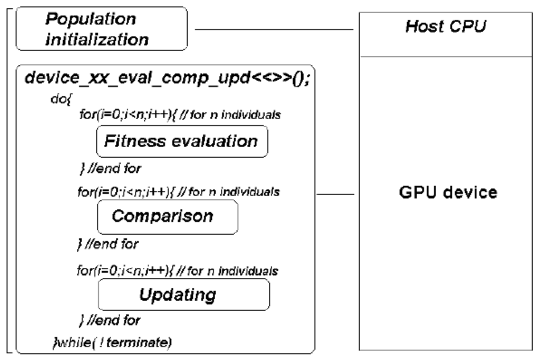
The following functional blocks can be observed in such metaheuristic:

• Population initialization. It initializes population's individuals in a random form.

• Fitness function evaluation.

• Comparison. It determines if an individual has better fitness that the best registered.

• Updating. Every individual updates its fitness following the specific algorithm rules.



```
device_xx_eval_comp_upd<<>>();
  do{
     for(i=0;i<n;i++){ // for n individuals
        Fitness evaluation
     } //end for
     for(i=0;i<n;i++){ // for n individuals
        Comparison
     } //end for
     for(i=0;i<n;i++){ // for n individuals
        Updating
     } //end for
  }while( ! terminate)
```

Fig. 3 Depicts The General Structure Of Sequential DE Algorithm.

Intentionally, in order to illustrate the straightforward parallelization of the code, the sequential functional blocks have been reorganized highlighting the loops that are in terms of the individuals number. Note that in the sequential implementation (see Fig. 2) all the functional blocks are necessarily executed on the host processor. By contrast, in our parallel implementation, called embedded, only the initialization module remains running on the host processor (see Fig. 3), since the kernel callis associated with the whole optimizing process that include fitness evaluation, comparison, and updating modules, all of them running on the GPU by mean of multiple threads (one by each individual), until a termination condition is reached.

Concerning the initialization module, the initialization of each of the individual's seeds (one seed per thread), used for generation of random numbers, is carried out on the host and remained out of the GPU. The above condition guarantees a good quality generation of random numbers into GPU for each thread and results, as is shown in the experimental results, in a different convergence behavior relative to the sequential algorithms. At this point, we must remark that the generation of random numbers was implemented in a different way for sequential and parallel codes, because it was particularly difficult to generate different random numbers within GPU when a single seed is used. Then, sequential codes kept the traditional way to generate random numbers based on a single seed, while the parallel codes used too many seeds as individuals in population as will be described further.

In Fig. 3 we can see that the device_xx_eval_comp_upd<<>>() kernel is invoked, where xx can be DE, depending on the implemented algorithm. For example, for our DE parallel implementation, firstly we declare an application Class that represents DE population. Then the principal optimization function is defined.

As is showed, the host can invoke the kernel function by mean of the wrapping function wrap_GPU_DE_Optimization(), that is defined within a CUDA file (in our case pde_emb_kernel.cu).

The kernel function is defined in the same CUDA file. Note that this code is parallelized in multiple threads, as many as individuals, which are distinguished by mean of a thread'sunique ID defined by blockId and threadId global structures assigned at runtime by the system [3]. In this code the fitness evaluation process is directly related to the objective function evaluation, for example for Generalized Griewank's (F03).

We must remark some practical considerations that should be taken into account to achieve functional implementation in parallelization of population-based algorithms on a multi-threading GPU:

• **Overhead**. The GPU presents an overhead due to latency in memory transferences between the host and GPU device. Because these transferences are relatively slow, any parallel implementation on a GPU must minimize their employment.

• **Synchronization**. Before any decision is taken, for example during the comparison and updating process, all the running threads must be synchronized in that point in order to obtain reliable information. Since in population based heuristics the individuals share information.

• **Contention**. This problem may occur if global variables are simultaneously revised by several threads. Appropriate precautions must be incorporated to deal with this problem. Specifically in the DE algorithm, this situation happens with the variable that contains the index to the global best.

• **Generation of random numbers**. This process may become a problem if the random numbers are generated within the GPU without an adequate strategy for the initialization of the seeds. Any call to the rand() function running on the GPU must generate different numbers for each thread and in any call. If the above condition is not provided, it could result in a no converging algorithm because of the poor diversity. In our parallel implementation one seed per individual (i.e. one seed per thread) is created and initialized by the host, and later each thread itself generates random number and updates its seed by mean (defined in pde_emb_kernel.cu, in our case file) but updates the seed after each call.

6. EXPERIMENTS AND RESULTS

The experiments were conducted on a PC with processor Intel Core Duo with Linux operating system, which is called 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 that allowed us to write parallel code fort he NVIDIA GPU in an almost straightforward way, as was described in the above section.

6.1 Experimental procedure

The goal of our experiments is assessing the performance of our parallel implementation for DE

, comparing it against the sequential DE version. The performance of DE algorithm was measured for a set of four well-known benchmark functions [17], varying both iterations and individuals number during the optimization of each objective function. Four multimodal functions were selected since they present significant optimization complexity [15]:

- F01 - Generalized Rosenbrock function.

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

$$-30 < x_i < 30$$

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

with $n = 30$ dimensions.

- F02 - Generalized Rastrigin's function.

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

$$-5.12 < x_i < 5.12$$

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

with $n = 30$ dimensions.

- F03 - Generalized Griewank's function.

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

$$-600 < x_i < 600$$

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

with $n = 30$ dimensions.

- F04 - Generalized Schwefel's function.

$$f_4(x) = \sum_{i=1}^{n}[-x_i \sin(\sqrt{|x_i|})]$$

$$-500 < x_i < 500$$

$$\min(f_4) = f_4(420.9687, \cdots, 420.9687) = -n \times 418.9829$$

with $n = 30$ dimensions.

All of these objective functions are multimodal, but F01 and F03 are separable functions while F02 and F04 are non separable ones, understanding that separable functions can be written as linear combinations of functions of individual variables.

The following two experiments were carried out to measure the performance of the parallel implementation:

•**Experiment 1**. For DE parallel implementation, performance measurements varying the iterations number. While the iterations number was variable (1000 to 31000, in steps of 2000 iterations), and the individuals were fixed to 128. The goal of this experiment is to compare the convergence curve and consumed time of the parallel algorithm against the sequential one in terms of the iterations number.

• **Experiment 2**. For DE parallel implementation, performance measurements varying the individuals number. While the individuals number was variable (64, 128 to 1024, in steps of 128 individuals), and the iterations were fixed to 15000. The goal of this experiment is to compare the convergence curve and consumed time of the parallel algorithm against the sequential version in terms of the individuals number. Note that in this experiment, individuals number was increased in multiples of 64, which results in the fact that our parallel implementations one in thread is created by each individual and because NVIDIA recommends the construction of threads blocks with size been multiple of 64, in order to better exploit the GPU resources [3].

For Algorithm 1, a DE/rand/1/bin algorithm was selected, parameter F was fixed at 0.6 for all the tested functions, while parameter CR was fixed depending on the function, as is recommended in [17]: 0.9 for F01 and F03 (non separable functions), but 0.0 for F02 and F04 (separable functions).

Each experiment was repeated 30 times for each benchmark function. Thus the average solution (specifically, the function valuation), its standard deviation, and the averaged consumed time in seconds was registered for each of the tested functions.

6.2 Performance metrics for parallel processing

Traditionally, to assess the performance of parallel implementations, the following metrics are defined:

• Computational cost and

• Speedup.

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

$$T = 1/C$$

Speedup S measures the reached execution time improvement and express how fast the parallel implementation is, compared with the implementation of reference:

$$S = T_{targ} / T_{ref}$$

where T_{targ} is the throughput of parallel implementation under study, and T_{ref} is the throughput of the sequential implementation in our case.

6.3 Experimental results

Because the goal is to compare the performance of DE parallel implementations running on the GPU the experimental results were registered for both algorithms (sequential and parallel) and for each tested function. In this section the observed behaviors for DE implementation, after varying iterations and individuals number, are commented.

Since it is well known that the convergence quality of a given population-based algorithm, during an optimization process, typically is very sensitive to its specific parameters (either CR and F for DE) and on the problem itself (i.e. objective function), determining were DE algorithm has a better convergence for each kind of problem is out of the scope of this work. Regardless DE convergence respect to a particular function, it is more interesting in determinate the effect of code parallelization itself and also the effect of varying individuals and iterations number on both the cost (i.e. consumed time) and the general shape of convergence curve.

All tested functions have an optimum value at zero except for F04. In order to have comparable plots, the plot of F04 was adjusted by just adding the optimum value (12569.4866 with 30 dimensions) to show a convergence curve referred to zero value.

6.3.1 Results of experiment 1: Experimental results varying iterations number, fixing individuals to 128, are plotted in Figures 4 to 9. In Fig. 4 we can see that, during F01 optimization, sequential (abbreviated as seq.) Parallel implementation of DE algorithm (emb. as short for embedded) has the best convergence compared with sequential one. The parallel DE algorithm converge first to the optimum around 3000 iterations, followed by sequential DE.

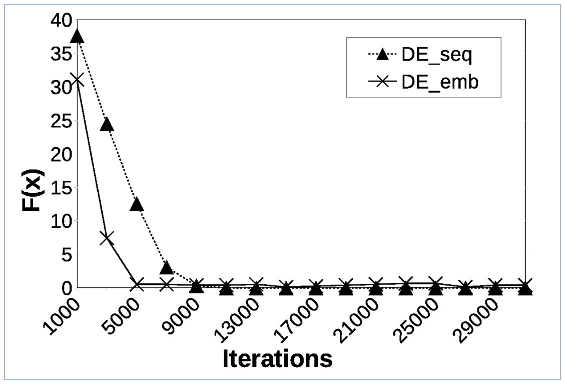


Fig. 4. Convergence For F01 Optimization Fixing Individuals To 128 And Varying Iterations

During F02 optimization, parallel implementation of DE algorithm has the best convergence, see Fig. 5. For this case, sequential implementation have a worse convergence compared with the parallel one.

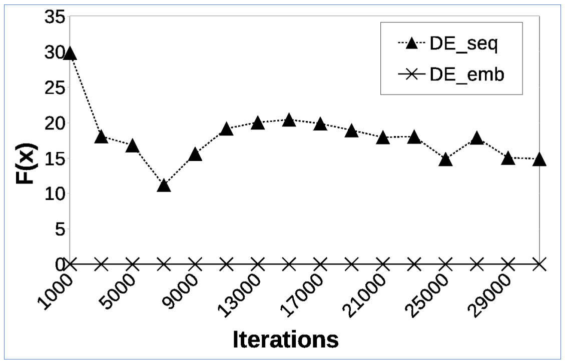


Fig. 5. Convergence For F02 Optimization Fixing Individuals To 128 And Varying Iterations

In Fig. 6 we can see that, during F03 optimization, parallel implementation of DE were trapped in a better local optimum than sequential implementation that is trapped in local optimums far from the global optimum.

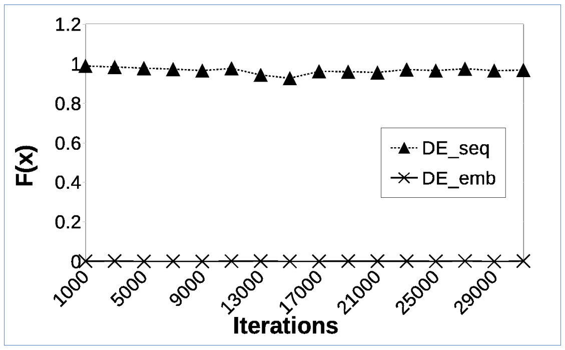


Fig. 6. Convergence For F03 Optimization Fixing Individuals To 128 And Varying Iterations

In Fig. 7 we can see that, during F04 optimization, parallel implementation of DE algorithm is trapped in a local optimum closer to the global optimum compared with the other sequential one, that are trapped in local optimums far from the global one. Specifically, F04 proved to be a hard optimization problem for all tested algorithms.

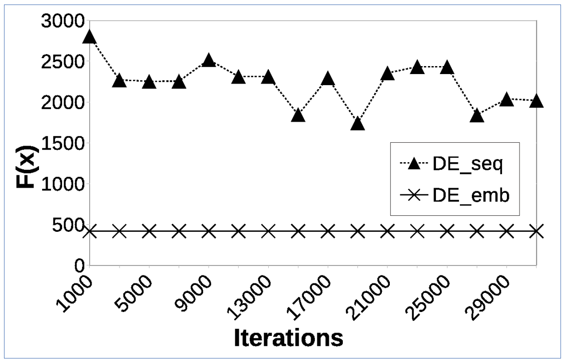


Fig. 7. Convergence For F04 Optimization Fixing Individuals To 128 And Varying Iterations

Concerning consumed time and reached speedup varying the iterations number, the experimental results are very similar for all the tested functions. For example, in Fig. 8 and Fig. 9 we can see the consumed time and speedup for F03 optimization.

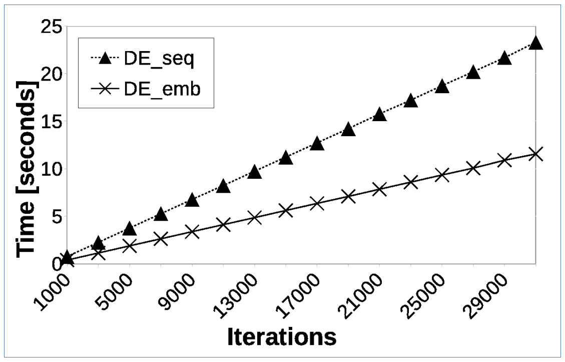


Fig. 8. Cost For F03 Optimization Fixing Individuals To 128 And Varying Iterations

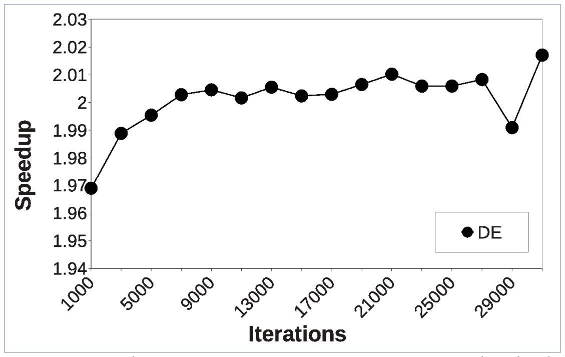


Fig. 9. Speedup For F03 Optimization Fixing Individuals To 128 And Varying Iterations

We can see a natural speedup introduced by the GPU that is almost invariant to variations in iterations number. The reached speedups with 128 individuals are very modest (2 for emb. DE), firstly because we fixed the population to allow value of 128 (it was observed that, in the proposed parallel implementations, the GPU improves the speedup when increasing the individuals number [2]) and, secondly, because in the proposed implementations are not exploited shared memory, coalesced memory instructions neither parallelizing arithmetic operations.

6.3.2 Results of experiment 2: Experimental results varying individuals number and fixing iterations to 15000 (after observing the convergence curves of Experiment 1 and observing that 15000 iterations gives the opportunity to reach a stable solution, either local or global optimum) are plotted in Figures 10 to 15. In Fig. 10, again, we can see that during F01 optimization parallel DE has the best convergence. On the other hand, sequential DE algorithm show an impoverishment as individuals are increased. In this case it is very interesting to note the opposite behaviors of parallel DE algorithm.

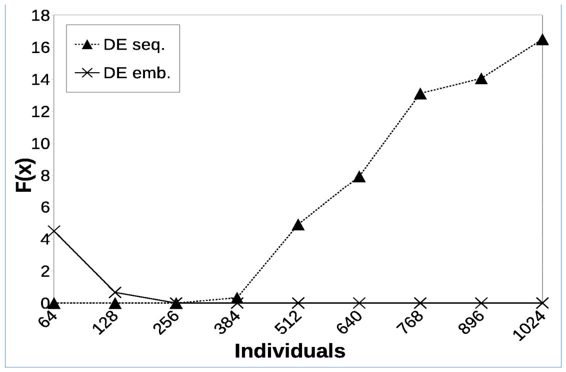


Fig. 10. Convergence For F01 Optimization Fixing Iterations To 15000 And Varying Individuals

In Fig. 11 again we can see that, during F02 optimization, parallel implementation of DE algorithm has the best convergence.

In this case, also sequential implementation has a worse convergence compared with the parallel one, but sequential DE has the slowest convergence. In Fig. 12 we can see that, during F03 optimization, parallel implementation of DE algorithm is trapped in a better local optimum than sequential implementation that is trapped in local optimums far from the global optimum. In Fig. 13 again we can note that, during F04 optimization, parallel implementation of DE algorithm is trapped in a local optimum closer to the global optimum compared with the other sequential and parallel ones, that are trapped in local optimums far from the global one.

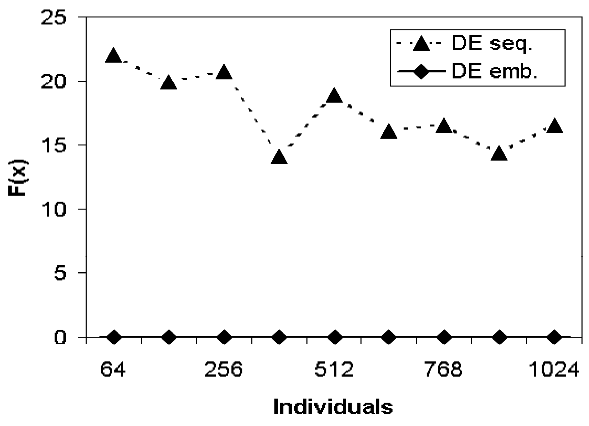


Fig. 11. Convergence For F02 Optimization Fixing Iterations To 15000 And Varying Individuals

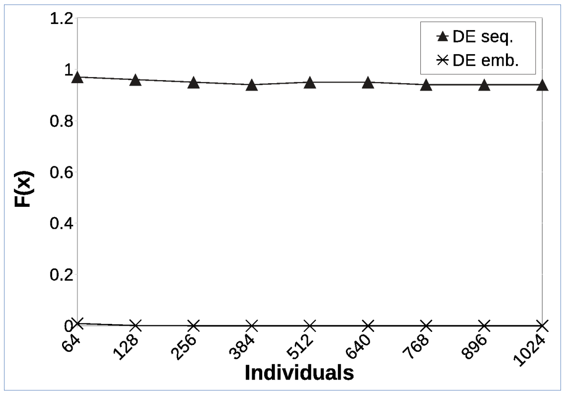


Fig. 12. Convergence For F03 Optimization Fixing Iterations To 15000 And Varying Individuals

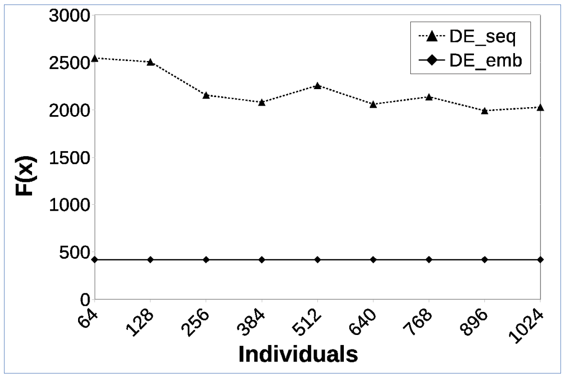


Fig. 13. Convergence For F04 Optimization Fixing Iterations To 15000 And Varying Individuals

Concerning consumed time and reached speedup varying the individuals number, the experimental results are very similar for all the tested functions. For example, in Fig. 14 and Fig. 15 we can see the consumed time and speedup for F03 optimization. In the proposed parallel implementation, at the beginning the GPU improves the speedup when the individuals are increased, but after a given point the speedup become almost constant. In fact, as the individuals are increased, the speedups are stabilized around five for DE algorithm, but these value can be improved as was described above.

6.4 Discussion of results

The above experiments confirmed that in the proposed parallel implementation the reached speedup is increased specifically when the individuals number is increased. According to the proposed implementation, this behavior is justified by the fact that increasing the individuals results in increasing the executed threads, and that give more chance to the GPU to make a better resource administration [3]. On the other hand, increasing the iterations simply forces the GPU to repeat more times the same processes. Thus, it is more useful to analyze the convergence curve and consumed time of the parallel implementations while the individuals number is varied than when the iterations number is varied. Concerning *speedup*, within the considered range, it must be clarified that the relatively modest values reported here may be improved with a more efficient code (by intensive use of shared memory, exploiting coalesced memory instructions and parallelizing arithmetic operations).

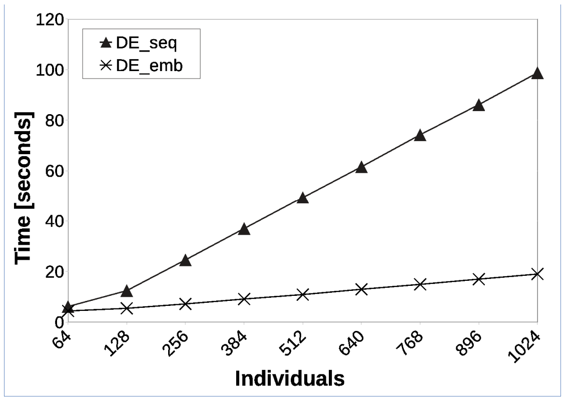


Fig. 14. Cost For F03 Optimization Fixing Iterations To 15000 And Varying Individuals

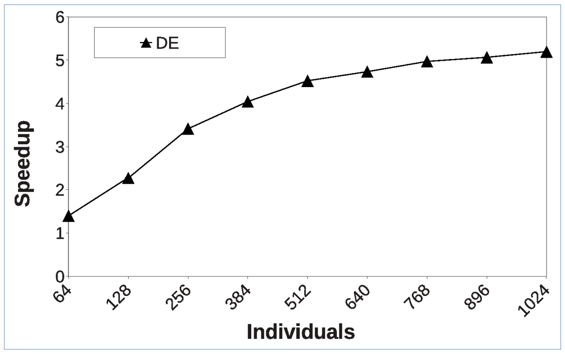


Fig. 15. Speedup For F03 Optimization Fixing Iterations To 15000 And Varying Individuals

Based on the fact that the proposed parallel algorithms are essentially the sequential ones after being migrated to GPU in a straightforward form, we can say that parallel algorithms and sequential ones are practically identical in all their functional modules with exception of the generation of the random numbers. In this sense, it is very interesting to note that the effect after a parallelization of a given heuristic on a GPU results in a natural speedup but additionally affects the convergence behavior in a significant manner. In our case this change in the convergence behavior is mainly because the different ways in which the random numbers are generated in parallel implementation with respect sequential ones. Since in the proposed parallelization the generation of random numbers into GPU is carried out in a truly and independently parallel way, individual by individual, the intrinsic behavior of the parallel implementation is different to the sequential one.

In general, the best performance was reached by the parallel DE algorithm during the optimization of all the tested functions. It is very interesting to note the change of convergence behavior between the sequential and parallel implementation of (i.e. DE). These experimental results show that with a parallel code and a NVIDIA GPU not only the execution time is reduced but also the convergence behavior to the global optimum may be changed in a significant manner with respect the original sequential code. Specifically, it imply that having a sequential algorithm that performs well on a given benchmark function does not guarantee that the same algorithm within a parallelized variant will be well behaved in the same problem and also imply, in counterpart, that having a sequential algorithm that performs badly on a given benchmark function does not guarantee that the same algorithm within a parallelized variant will be badly behaved in the same problem (as is clear in Fig. 11 and Fig. 12 for DE when optimizes F02 and F03).

7. CONCLUSION AND FUTURE WORK

In this paper, a parallel version of DE algorithm, is implemented on a multithreading GPU using CUDA as the model of parallel programming, were presented. Some insights about the proposed parallel implementation for population-based algorithms on a GPU were given. Specifically, an approach called embedded, where there is one thread per individual, was used. It must be noted that the proposed approach is not strictly any of the well known parallel models (global, island, nor diffusion) traditionally used in these cases, but is similar to the diffusion one but there is one thread instead of one processor per individual. Regardless the natural speedup introduced by the GPU, it is shown that the proposed parallel implantations may have different behavior, compared to sequential ones, as a result of the specific way in which the random numbers are generated within GPU.

The experimental results showed that parallel DE algorithm has a better behavior optimizing the most of our set of well know benchmark functions, with significant dimensionality of 30 and algorithm's parameters specified. It is notable that DE, using the appropriate values for CR and F parameters that depend on the problem, may converge to the global optimum with few particles as 64 and few iterations as 3000.

Future research may be focused on the following three points:

(1) Apply the whole multithreading GPU capacities, including intensive use of shared memory, exploiting coalesced memory instructions and parallelizing arithmetic operations, to get a powerful high parallelized DE algorithm.

(2) Test more fitness functions with bigger dimensionality.

(3) Employ these parallel problem Algorithms to a complex real world optimization**.**

REFERENCES

[1] Owens, J.D. and Luebke,D. and Govindaraju, N. and Harris, M. And Kr¨uger, J. and Lefohn, A.E. and Purcell, T.J., "*A Survey of General purpose Computation on Graphics Hardware*", Computer Graphics Forum, 2007

[2] Laguna-Sánchez G., Olguín-Carbajal M., Cruz-Cortés N., Barrón Fernández R. and Alvarez-Cedillo J., "*Comparative Study of Parallel Variants for a Particle Swarm Optimization Algorithm Implemented on a Multithreading GPU*", Journal of Applied Research and Technology (JART), 2009

[3] NVIDIA Corporation, "*CUDA Computer Unified Device Architecture Programming Guide*", Version 6.5, PG-02829-001 v6.5 2014

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

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

[6] Liaoa T., Stutzlea T., Montes de Oca M. A., Dorigo M. 2*A United Ant Colony Optimization Algorithm for Continuous Optimization*". IRIDIA, Universite Libre de Bruxelles. IRIDIA Technical Report Series, Technical Report No.TR/IRIDIA/2013-002, ISSN 1781-3794., 2013

[7] Olguín-Carbajal M., Herrera Lozada J.C., Arellano Verdejo J., Barron Fernandez R. and Taud H., "*Micro Differential Evolution Performance Empirical Study for High Dimensional Optimization Problems*", Lecture Notes in Computer Science LNCS 8353, Springer 2014

[8] Olguin-Carbajal M., Alba E. and Arellano Verdejo J., "*Micro Differential Evolution with local search for High Dimensional Problems*", 2013 IEEE Congress on Evolutionary Computation, 2013

[9] Mukeherjee R., Debchoudhury S., kundu R., Das S. and Suganthan P.N., "*Adaptive Differential Evolution with locality Based Crossover for Dynamic Optimization*", 2013 IEEE Congress on Evolutionary Computation, 2013

[10] Boloufe Rohler A., Estevez Velarde S., Piad Morffis A. Chen S. And Montgomery J, "*Differential Evolution with Thresheld Convergence*", 2013 IEEE Congress on Evolutionary Computation, 2013

[11] Schutte, J.F. and Reinbolt, J.A. and Fregly, B.J. and Haftka, R.T. and George, A.D., "*Parallel Global Optimization with the Particle Swarm Algorithm*", International Journal for Numerical Methods in Engineering, 2003

[12] Ma, H.M. and Ye, C.M. and Zhang S., "*Research on Parallel Particle Swarm Optimization Algorithm Based on Cultural Evolution for the Multi-level Capacitated Lot-sizing Problem*", IEEE Control and Decision Conference, 2008

[13] Harding, S. and Banzhaf, W., "*Fast Genetic Programming on GPUs*", 10th European Conference on Genetic Programming, Lecture Notes in Computer Science, Editor Ebner, M. 2007

[14] Li, J.M. and Wang, X.J. and He, R.S. and Chi, Z.X., "*An Efficient Fine-grained Parallel Particle Swarm Optimization Method Based on GPU-acceleration*", International Journal of Innovative Computing, Information and Control, 2007

[15] Mzoughi, A. and Lafontaine, O. and Litaize, D., "*Performance of the Vectorial Processor VECSM2* Using Serial Multiport Memory*", 10th International Conference on Supercomputing, 1996

[16] Wolpert, D.H. and Macready, W.G., "*No Free Lunch Theorems for Optimization*", IEEE Transactions on Evolutionary Computation, 1997

[17] Storn, R. and Price, K.V., "*Differential Evolution a simple and efficient heuristic for global optimization over continuous spaces*", Journal of Global Optimization, 1997

[18] Swagatam Das and Ponnuthurai Nagaratnam Suganthan, "*Differential Evolution: A Survey of the State-of-the-Art*", in IEEE Transactions on Evolutionary Computation, Vol. 15, No. 1, February 2011.

[19] Mezura-Montes, E. and Velázquez-Reyes, J. and Coello-Coello, C.A., "*A Comparative Study of Differential Evolution Variants for Global Optimization*", GECCO, 2004

[20] Belal, M. and El-Ghazawi, P., "*Parallel Models for Particle Swarm Optimizers*", International Journal of Intelligent Computing and Information Sciences, IJICIS, 2004