# A Parallel Version of the JADE Algorithm using GPUS

Adriana Mexicano[1], Jesus C. Carmona[1], Nelva N. Almaza[2], Lilia Garcia[1], Francisco Arguelles[1]
[1]Division de Estudios de Posgrado e Investigacion, Tecnologico Nacional de Mexico, Tamaulipas
[2]Division de Estudios de Posgrado e Investigacion, Tecnologico Nacional de Mexico, Tlalnepantla
{mexicanoa@gmail.com} {jcarmonafrausto@gmail.com.mx}

***ABSTRACT:*** *This work presents a parallel implementation of JADE: Adaptive Differential Evolution With Optional External Archive, using the Compute Unified Device Architecture (CUDA), in order to reduce the execution run-time of the algorithm. The algorithm was tested using the well-known function Sphere and the execution run time was compared against its sequential version. The results were measured in terms of "Speed-up" and they show that the execution run-time can be reduced significantly by the use of CUDA, this benefit can be observed better when working with large amounts of data. However, not necessarily the population with more data reaches the best performance.*

**Keywords:** Optimization, JADE, CUDA, Parallelization

**Received:** 12 June 2021, Revised 17 August 2021, Accepted 28 August 2021

**DOI:** 10.6025/dspaial/2022/1/1/1-10



## 1. Introduction

Evolutionary Algorithms (EA) are search and optimization procedures inspired by the biological world, they are characterized by imitating adaptive processes of natural systems and are based on the survival of the best individual. They work on populations of solutions that evolve from generation to generation through genetic operators adapted to the problem. One of them is the Differential Evolution algorithm proposed by Storn and Price in 1995, which has received much attention within the field of evolutionary computing because it has proven to be a simple, precise, and robust optimization method [9]. Evolutionary Algorithms have proven to be a good alternative to classical optimization methods, providing solutions to practical problems in various areas of knowledge such as design, finance, bioinformatics, pattern recognition among others, an EA is used given the enormous number of possible configurations and the need for an efficient search. As an example of the application of these methods to solve real problems we can mention the optimization of the duration cycles of the traffic lights [15] and the control of the flow of water that passes through a turbine to generate energy [13] among others.

The Differential Evolution algorithm (DE) [11] is used as an optimization method belonging to the evolutionary computation category, applied in solving complex problems. Like other algorithms in this category, DE maintains a population of candidate solutions, which recombine and mutate to produce new individuals that can be chosen according to the value of their perfor-

mance function [1]. Which characterizes ED is the use of test vectors, that compete with individuals in the current population in order to survive.

A limitation that is immersed when using sequential versions of DE algorithms when it is necessary to solve real optimization problems is their execution time. In works such as [4], [7], [12], [3] parallel versions of the algorithms have been implemented to reduce this inconvenience. Therefore, the use of Graphical Processing Units (GPUs) used as a means of parallelization are gaining more and more importance due to their great computing potential, scalability, and low cost. There are different architectures that show the advantages of differential evolution parallelization (MPI, CUDA, etc.), among which the CUDA architecture stands out, because is a programming model and a calculation architecture [14] that presents a good cost benefit relation.

According to specialized literature, the JADE algorithm is a DE algorithm widely used in solving real problems and to date we have not found any parallelized version using the technology of GPUs. Therefore, in this work the parallelized version of the JADE algorithm proposed by Zhang and Sanderson in 2009 [18] was implemented to reduce its execution time using the CUDA architecture.

## 2. JADE: Adaptive Differential Evolution with Optional External File

JADE is an adaptive differential evolutionary algorithm with optional external file proposed by Zhang and Sanderson [16] [6], which implements a neighborhood-based mutation strategy and an optional external file used to improve the performance of the DE.

The steps of the algorithm are as follows:

### 2.1 Step 1. Initialization

JADE follows the basic procedure of differential evolution. The initial population $\{x_{i,0} = (x_{1,i,0}, x_{2,i,0}, \ldots, x_{D,i,0}) | i = 1, 2, \ldots, NP\}$ is generated randomly according to a uniform distribution $x_j^{low} \leq x_{j,i,0} \leq x_j^{up} | j = 1, 2, \ldots, D$, where $D$ is the dimension of the problem, NP is the population size, $x_j^{low}$ and $x_j^{up}$ define the upper and lower limits of the j-th decision variable. After initialization, JADE enters in a loop of evolutionary operations (mutation, crossing, and selection) and parameter adaptation operations.

### 2.2 Step 2. Mutation

Equation 1 shows the mutation vector generated by DE / current-to-pbest, with file:

$$V_i^G = X_i^G + F_i \cdot (X_{best,p}^G - X_i^G) + F_i \cdot (X_{r1}^G - \tilde{X}_{r2}^G) \quad (1)$$

where $X^G{}_{best,p}$ is randomly selected as one of the best 100p individuals from the current population with $p \in (0, 1]$. While, $X^G{}_i$ and $X^G{}_{r1}$ are random individuals in the current population $P$, respectively. $\widetilde{X}^G{}_{r2}$ is randomly selected from the union of $P$ and file $A$. A is a set of solutions from previous generations and its number of individuals is not greater than the size of the population. In each generation, the mutation factor $F_i$ and the probability of crossing $C_{ri}$ of each individual $X_i$ are, respectively, dynamically updated according to a Cauchy distribution with mean $\mu_F$ expressed in equation 2 and a normal distribution with mean $\mu_C$, described in equation 3:

$$F_i = rand\,c_i(\mu_F, 0.1) \quad (2)$$

$$C_{ri} = rand\,n_i(\mu_{Cr}, 0.1) \quad (3)$$

The two proposed parameters are initialized as 0.5 and at the end of each generation they are generated with equations 4 and 5:

$$\mu_F = (1 - c) \cdot \mu_F + c \cdot mean_L(S_F) \quad (4)$$

$$\mu_{Cr} = (1 - c) \cdot \mu_{Cr} + c \cdot mean_A(S_{Cr}) \quad (5)$$

where $c$ is in a range (0, 1); $S_F$ and $S_{Cr}$ indicate the set of all the mutation factors and probabilities of successful crosses in the

generations; $mean_A(\bullet)$ denotes the usual arithmetic mean, and $mean_L(\bullet)$ is the Lehmer mean, which is defined in equation 6:

$$mean_L(S_F) = \frac{\sum_{i=1}^{|S_F|} F_i^2}{\sum_{i=1}^{|S_F|} F_i} \tag{6}$$

### 2.3. Step 3. Cross

After the mutation, the cross-binomial operation $u_{i,g} = (u_{1,i,g}, u_{2,i,g}, \ldots, u_{D,i,g})$ forms the vector test as shown in equation 7:

$$u_{j,i,g} = \begin{cases} V_{j,i,g} \; if \; rand_j(0,1) \lesssim RC_i \; o \; j = j_{rand} \\ X_{j,i,g} \; otherwise \end{cases} \tag{7}$$

where $rand_j(0, 1)$ is a random number in the interval (0, 1) and is generated for each $j$, $j_r$ and = $randint_i(1, D)$ is a randomly chosen integer between 1 and Dand is generated for each $i$, and the crossover probability $CR_i \in [0, 1]$, corresponds approximately to the average fraction of the vector components that are inherited from the mutation vector.

### 2.4. Step 4. Selection

The selection operation chooses the best between the parent vector $x_{i,g}$ and the test vector $u_{i,g}$ according to the values of the evaluation function. For example, if we have a minimization problem, the selected vector is defined as shown in equation 8:

$$x_{i,g+1} = \begin{cases} U_{i,g} \; if \; f(U_{i,g}) < f(X_{i,g}) \\ X_{i,g} \; otherwise \end{cases} \tag{8}$$

The operation is called correct update if the test vector $u_{i,g}$ is better than that of the parents $x_{i,g}$, that is, the improvement or progress of the evolution $\Delta_{i,g} = f(x_{i,g}) - f(u_{i,g})$ is positive. Consequently, the control parameters $F_i$ and $CR_i$ used to generate $u_{i,g}$ are called the successful mutation factor and successful crossover probability, respectively.

## 3. Graphic Processing Units

Driven by market demand for high-definition 3D interactive graphics, Graphic Processor Units (GPUs) have evolved into highly parallel architectures with many processors and a large computing power [6].

The GPUs are specialized microprocessors that allow the acceleration of the representation of the graphics from the Central Processing Unit (CPU) to lighten its workload, that is, while the GPU is responsible for the representation of the graphics, the CPU is doing other calculations. Current GPUs are more efficient in manipulating graphics in a computer, they have a highly parallel architecture that has been shown to be more efficient than CPUs for the implementation and execution of certain algorithms [10]. A GPU can be present in a PC in the form of a video card or at present it can even be the base card.

In 2006 NVIDIA introduced a new general-purpose parallel computing architecture called CUDA (Compute Unified Device Architecture) capable of taking full advantage of the capabilities of GPUs in order to solve complex computational problems in a more efficient way. CUDA enables developers to use the C language as a high-level implementation language.

CUDA allows C code to be translated almost directly to the device. This model unifies the CPU (host) and the GPU (device) in a heterogeneous computing system, to get the best out of both devices, it contains three types of functions the Host Function which are called and executed in the CPU, the Kernel Function which are called on the host and executed on the device; and the third is the Device Function which are called and executed on the GPU. CUDA presents in its structure the concepts of “thread”, ”block” and ”grid” and the threads are organized in blocks and can be upto 3 dimensions, and the blocks are organized in grids which can also be 3 dimensions. Each block of threads is executed in a Streaming Multiprocessor (SM). Each SM allows a number of blocks to be executed, once the maximum block capacity per SM is reached, the remaining blocks have to wait to be executed in the SM.

## 4. Algorithm Parallelization

This section describes the parallelization of the algorithm according to its previously described phases:

### 4.1. Initialization

The population is generated randomly within the host for each generation (g), the population has a size $NP(i = 1, 2, 3, ..., NP)$, and each individual in a population has $D$ characteristics ($j = 1, 2, 3, ..., D$); therefore a population is represented by a matrix (Figure
1), where the rows represent each individual and the columns their characteristics, it is important to note that the matrix coincides with the way the population data is stored in the host, which facilitates the manipulation of the data through the phases.

For example, when creating a population of $NP = 2$ with $D = 5$ characteristics, with values in $x \in [-100, 100]$, a matrix $x$ is obtained:

$$x = \begin{pmatrix} -45 & 72 & 38 & -64 & -54 \\ 1 & -82 & 12 & 89 & 96 \end{pmatrix}$$

### 4.2. Function Evaluation

Once the initial population or first generation ($g$) is generated, it is sent to the kernel to carry out the function evaluation, Taking the example of the population matrix from the previous phase when evaluating the Sphere 9 function:

$$f(x) = \sum_{i=1}^{D} Z_i^2 \tag{9}$$

Where $x$ is the population matrix, or are values in a range of [-100,100], these values are provided in the CEC [17] and $z$ is the result of the difference between the matrix $x$ minus the matrix $o$ $z = x - o, x = [x_1, x_2, \ldots, x_D]$, and $o = [o_1, o_2, \ldots, o_D]$. Following matrix $z$ and $o$ are presented:

$$o = \begin{pmatrix} -39.311 & 58.899 & -46.322 & -74.651 & -16.799 \\ -39.311 & 58.899 & -46.322 & -74.651 & -16.799 \end{pmatrix}$$

$$z = \begin{pmatrix} -6 & 13 & 84 & 11- & 37 \\ 40 & -141 & 58 & 164 & 113 \end{pmatrix}$$

Applying the Sphere function described in equation 9 the vector with the best $f(x)$ values is obtained:

$$f(x) = (8811.55, 64384.92)$$

The best value is taken and based on its index within the vector, the individual from the population matrix $x$ is taken, in this case it is individual 1, which is the one that is closest to the objective function creating the vector *xbest*:

$xbest = (-45, 72, 38, -64, -54)$

Figure 2 shows how the population data is arranged within the grid, in the form of a matrix, where each data is evaluated independently. The size of the grid is calculated by applying equation 10:

$$Gridsize = \frac{NP - 1}{blockWidth + 1}, \frac{D - 1}{blockWidth + 1} \tag{10}$$

where $NP$ is the size of the population, $D$ indicates the number of dimensions or characteristics, and blockWidth is the length of the block. For example, when using a population of $NP = 100$, a dimension with $D = 10$ and a length of the block width Block = 32, since the maximum capacity of the hardware is 1024 threads, blocks with dimensions of 32x32 threads were used; substituting the values in equation 10, the following result is obtained:

$$Gridsize = \frac{100 - 1}{32 + 1}, \frac{10 - 1}{32 + 1} = 3, 1$$

Each of the population data is evaluated by a thread, in this phase the objective function is applied to generate a vector with the objective values:

### 4.3. Mutation

Figure 3 shows how the population changes when applying the mutation process. Once the individuals have been evaluated in phase 2, the "DE/current " to "pbest" mutation strategy is implemented, which consists of evaluating each individual. To determine its degree of aptitude, that is, whether or not it approaches the solution, applying the mutation strategy of equation 11 to the example, we have the mutation matrix *v*:

$$V = x + F \cdot (xbest - x) + F \cdot (r1 - r2) \tag{11}$$

where *x* is the population matrix, the mutation factor $F = 0.5$, the vector with the best individual *xbest, r*1 and *r*2 are random individuals from the current population.

$$v = \begin{pmatrix} -68 & 149 & 51 & -140.5 & -129 \\ -45 & 72 & 38 & -64 & -54 \end{pmatrix}$$

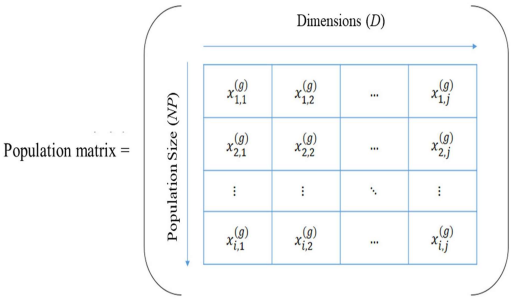


Figure 1. Population initialization

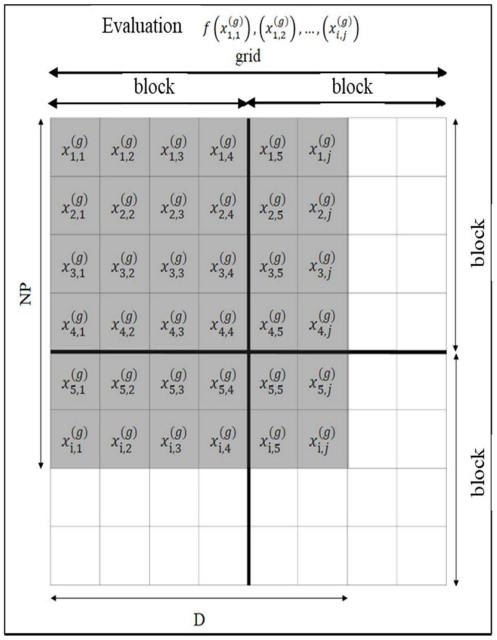


Figure 2. Function evaluation

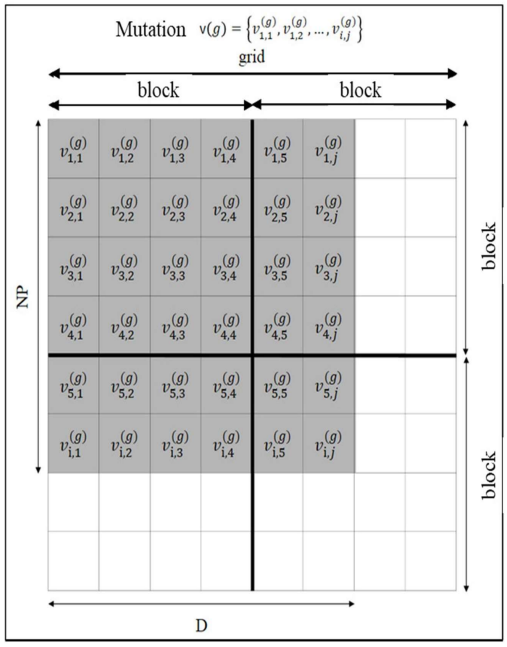


Figure 3. Mutation

### 4.4. Crossing

Once the mutation phase has been applied, a recombination of the population generated in the matrix *x* and the mutation matrix *v* is carried out, forming a test matrix *u*, as shown in equation 12:

$$u = \begin{cases} v \; if \; rand(0,1) \leq CR \; or \; j = jrand \\ x \; otherwise \end{cases} \tag{12}$$

where *rand*(0, 1) is a vector of random numbers between 0 and 1, the probability of crossing *CR* = 0.5 and the vector j are the values of the indices of the population matrix; the crossing function loads the characteristics of the individuals according to the conditions of equation 12 where if the values of the rand vector are greater than or equal to the probability of crossing *CR* or if the indices of vector *j* are equal to the random indices of the *jrand* vector then the value of the mutation matrix *v* is taken, otherwise the characteristic of the population matrix *x* is loaded:

$$u \begin{pmatrix} -68 & 72 & 51 & -64 & -54 \\ 1 & 72 & 38 & -64 & 96 \end{pmatrix}$$

Figure 4 shows the change within the grid when performing the crossing phase:

**4.5. Selection**

The selection operation chooses the best individual between the parent vector *x* and the test vector *u* generated in the cross operation; according to the values of the evaluation function, as shown in equation 13 to generate the population of the new generation resulting in the matrix *x*2:

$$x2 = \begin{cases} u \; if \; f(u) \leq f(x) \\ x \; otherwise \end{cases} \tag{13}$$

where it takes the first individual from the population matrix and the second is obtained from the cross matrix:

$$x2 = \begin{pmatrix} -45 & 72 & 38 & -64 & -54 \\ 1 & 72 & 38 & -64 & 96 \end{pmatrix}$$

Figure 5 shows the selection process within the grid.

**5. Experimentation**

**5.1. Test Environment**

The test environment was established under the following conditions: A laptop with an Intel core i7 processor, 6 GB of RAM memory, 1 TB hard disk, Windows 10 operating system was used. A GeForce GT 650M graphics card with 384 CUDA cores, processor clock 900 MHz.

The Sphere test function was used during the evaluation of the algorithms, which was extracted from the CEC 2005 (Congress on Evolutionary Computation) [17]. The Sphere function is a well-known simple function with a single global minimum. The configuration used for the JADE algorithm in its parallel and sequential version was the following: *Mutation factor (F)* = 0.5, Probability of crossing (*Cr*) = 0.5, Generations (*G*) = 200.

Regarding the population size used during the experimentation, five population sizes 30 *D* x 100 *NP*, 100 *D x* 100 *NP*, 100 D x 1,000 NP, 100 D x 10,000 NP and 100 D x 20,000 NP were used. These five population sizes were defined according to state-of-the-art tests [8]. Each of the five configurations were executed in the parallel and sequential version; and each version was executed with the three evaluation functions mentioned previously. It is important to mention that all the experiments were executed 30 times.

**5.2. Evaluation of the Sequential against the Parallel JADE using the Sphere Function**

This section shows a comparison between the performance of the sequential JADE algorithm and the parallel version proposed in this work. The results are expressed in terms of execution times (seconds) and acceleration factor known as “Speed “ up”.

Speed-up [2] represents the performance growth ratio between sequential execution time and parallel execution time. The expression 14 corresponds to the “Speed “ up” calculation, where S represents the “Speed “up”, while CPU Time and GPU Time

correspond to the execution times according to the hardware where the experiment is carried out. If the obtained accuracy factor is less than or equal to one (S1), it means that no benefit is obtained from the parallel version.

$$S = \frac{CPUTime}{GPUTime} \tag{14}$$

Table 6 presents the results of the experimentation using the evaluation function ”Sphere”, where the first column shows the size of the population used in the experiment, in the column CPU Time and GPU Time the average times are shown of 30 executions carried out in their respective versions. The column “Speedup” shows the acceleration that the parallel version presented against the sequential one. As an example, we can interpret the penultimate row of the table as follows, for a population of 100 D x 10,000 NP the sequential version of the algorithm took 542.262100s, while the parallel version took 8.6684s, therefore, the version parallel is 62.5561 times faster. It is worth mentioning that a total of 300 tests were performed using the “Sphere” test function as a result of executing the 10 experiments presented in Table 6 30 times.

In Figure 6 the information reported in Table 1 is presented, where the “*x*” axis represents the different population sizes used in the experimentation, the “*y*” axis shows the average latency times in the experiments carried out, particularly due to Since the experiment with the population 100 D x 20,000 NP yields a very high time in the CPU (1866.1677s), it was necessary to scale the “y” axis of the graph, so that small values can be visualized in the first experiments and a High value in the last experiment, as observed in the “y” axis of Figure 21, the time of the experiments ranges from 0.01s to 1000s; the blue bars correspond to the execution times of the parallel version and the gray bars correspond to the execution times of the sequential version.

It is noteworthy that in the 100 D x 10,000 NP experiment using the “Sphere” test function, the parallel version of the JADE algorithm is 62 times faster, as can be seen in Figure 21, to interpret the graph it is necessary to take into account the scaling of the “*y*” axis.

The performance of the parallel version increases as the amount of data that must be processed increases. Also, the “Speed-up” values vary according to the test function used. Due to the overhead of making a kernel function call, the parallel version does not have an advantage over the sequential version when processing small amounts of data [10].

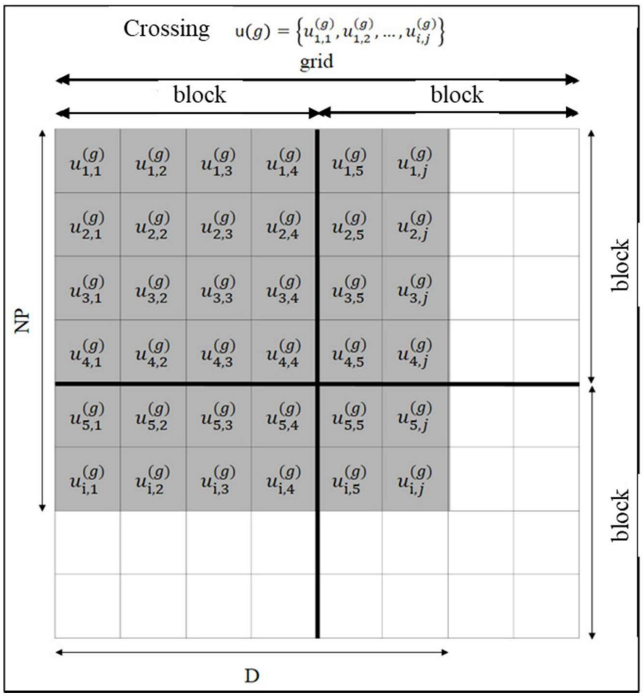


Figure 4. Crossing

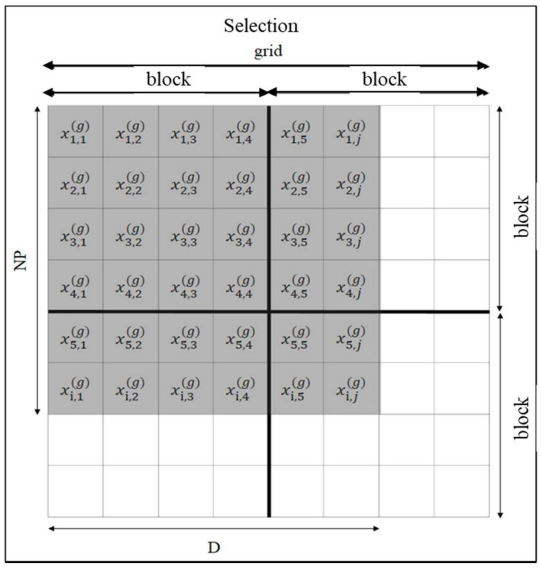


Figure 5. Selection

| D x NP | CPU Time (s) | GPU Time (s) | Speed-up(S) |
|---|---|---|---|
| 30 x 100 | 0.079067 | 0.2366 | 0.3341 |
| 100 x 100 | 0.1976 | 0.3973 | 0.4973 |
| 100 x 1,000 | 1.634867 | 13.5976 | 0.6277 |
| 100 x 10,000 | 542.2621 | 8.6684 | 62.5561 |
| 100 x 20,000 | 1866.1677 | 50.5901 | 36.8879 |

Table 1. Comparison between CPU Time against GPU Time using the Sphere function

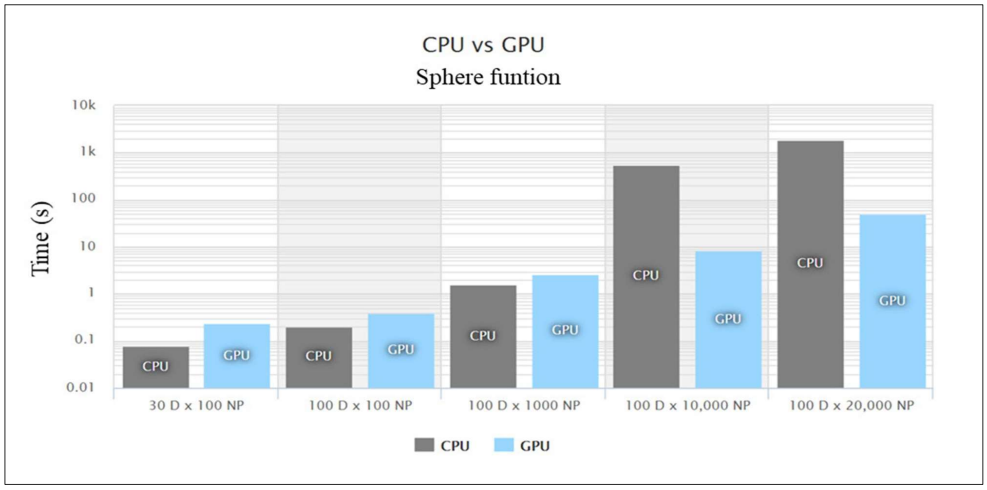


Figure 6. Comparison of CPU Time against GPU Time using the Sphere function.

The performance of the parallelized JADE algorithm proposed in this work in large populations is noteworthy. According to [5] it is advantageous to have a large population because there is a positive relationship between population size and population variation. If the population size is small, the algorithm has a probability of converging prematurely, that is, the population variation occurs abruptly and local minima are reached; or have a stagnation, this occurs when the population remains diverse, but does not converge, therefore the optimization process does not progress. In most cases, large populations are more stable.

## 6. Conclusion

The CUDA architecture of NVIDIA cards allows to efficiently solve problems that can be expressed under the scheme ”one instruction, multiple data” (SIMD, Single Instruction, Multiple Data). The differential evolutionalgorithm adapts to the SIMD scheme, since the stages of this algorithm (mutation, crossing and selection) execute the same instruction on a set of data (population).

In particular, the parallel version of the JADE algorithm, proposed in this work, showed a performance that widely exceeded, in terms of execution times, the sequential version of the JADE algorithm, resulting in an increase in the acceleration factor “Speed “ up”.

The performance evaluation of the parallel version of the JADE algorithm was carried out as follows, using the Sphere test function and five population sizes 30 D x 100 NP, 100 D x 100 NP, 100 D x 1,000 NP, 100 D x 10,000 NP, and 100 D x 20,000 NP, due to the randomness characteristics of the algorithms, each one (sequential and parallel) was executed 30 times, using each of the previously mentioned population sizes.

The results show that the parallel version of the JADE algorithm achieves the best computation times and the best values for the acceleration factor when using the 100 D x 10,000 NP setting. The parallel version was observed to be 62.5 times faster than the sequential version.

It can be concluded that the performance of the parallel version increases in relation to the amount of data processed, the larger the size of the dimensions (D) and the size of the population (NP), a considerable benefit will be obtained from the parallel version.

## References


[1] Arellano, J., Guzman, A., Godoy, S., and Barron, R. (2016). Efficiently finding the optimum number of clusters in a dataset with a new hybrid differential evolution algorithm: Dela. *Soft Computing* 20 (3) 895–905.

[2] Conte, D., Dambrosio, R., and Paternoster, B. (2016). Gpu-acceleration of waveform relaxation methods for large differential systems. *Numerical Algorithms,* 71 (2) 293–310.

[3] Davendra, D., Gaura, J., Bialic-Davendra, M., Senkerik, R. (2012). Cuda based enhanced differential evolution: A computational analysis. *In*: Proceedings 26th European Conference on Modelling and Simulation. Univerzita Tomase Bati ve Zline, 399–404.

[4] De Veronese, L., Krohling, R. A. (2010). Differential evolution algorithm on the gpu with c-cuda. In: Proceedings of the IEEE Congress on Evolutionary Computation CEC (2010), *IEEE Xplorer*, 1–7.

[5] Hu, D., Harding, S., Banzhaf, W. Variable population size and evolution acceleration: a case study with a parallel evolutionary algorithm. Genetic Programming and Evolvable Machines 11 (2) 205–225.

[6] Jimenez, C. O. F. A gpu-based parallel object kinetic monte carlo algorithm for the evolution of defects in irradiated materials. *Computational Materials Science* 113 (4) 178–186.

[7] Kr Nomer, P., Platos, J., Sn Lasel, V., and Abraham, A. Many-threaded differential evolution on the gpu, natural computing series. In Massively Parallel Evolutionary Computation on GPGPUs (2013), Springer-Verlag, 1–7.

[8] Lee, C., Yao, X. (2004). Evolutionary programming using mutations based on the levy probability distribution. *Evolutionary Computation*, 8 (1) 1–13.

[9] Lee, K. Y., El-Sharkawi, M. A. (2008). Modern Heuristic Optimization Techniques: Theory and Applications to Power

Systems. Wiley-IEEE Press, New Jersey.

[10] NVIDIA. Nvidia cuda c++ programming guide, 2021.

[11] Price, K. V., Storn, R. M., Lampinen, J. A. (2005). Differential Evolution: A Practical Approach to Global Optimization. Springer-Verlag, New York.

[12] Qin, A., Raimondo, F., Forbes, F., Ong, Y. (2012). An improved cuda-based implementation ofdifferential evolution on gpu. In Proceedings of the 14th annual conference on Genetic and evolutionary computation (2012). *Association for Computing Machinery,* 991–998.

[13] Regulwar, D., Choudhari, S., Raj, P. (2010). Differential evolution algorithm with application to optimal operation of multipurpose reservoir. *Journal of Water Resource and Protection*, 2 (6) 560–568.

[14] Rivera, I., Vargas-Lombardo, M. (2012). Principios y campos de aplicaci Lon en cuda programaci Lon paralela y sus potencialidades. *Nexo Revista Cientifica*, 25 (2) 39–46.

[15] Sanchez, J., Galan, M., Rubio, E. (2004). Genetic algorithms and cellular automata: A new architecture for traffic light cycles. *In Proceedings of the 2004 Congress on Evolutionary Computation*, *IEEE Xplorer*, 1668–1674.

[16] Sanderson, A. C., Zhang, J. (2009). Adaptive Differential Evolution: A Robust Approach to Multimodal Problem Optimization. Springer Publishing Company, New York, 2009. *Evolutionary Computation,* 13 (5) 945–958.

[17] Suganthan, P. N., Hansen, N., Liang, J. J., Deb, K., Ping Chen, Y., Auger, A., and Tiwari, S. (2005). Problem definitions and evaluation criteria for the cec 2005, special session on realparameter optimization.

[18] Zhang, J., Sanderson, A. C. (2009). Jade: adaptive differential evolution with optional external archive. *IEEE Transactions on Evolutionary Computation*, 13 (5) 945–958.