The Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Jaya optimization algorithm with GPU-acceleration

A. Jimeno-Morenilla · J.L. Sánchez-Romero · H. Migallón · H. Mora-Mora

Received: date / Accepted: date

Abstract The purpose of optimization methods consists in look for an optimal value given a specific function within a constrained or unconstrained domain. These methods are useful for a wide range of scientific and engineering applications. Recently, a new optimization method called Jaya has generated a growing interest due to its simplicity and efficiency. In this paper we present the Jaya GPU based parallel algorithms developed, we analyze both the parallel performance and the optimization performance using a well-known benchmark of unconstrained functions. The results indicate that the parallel Jaya implementation achieves a significant speed-up for the whole benchmark functions, obtaining speed-ups of up to $190x$, without disturbing the optimization performance.

Keywords Jaya, optimization, parallelism, GPU, CUDA

This research was supported by the Spanish Ministry of Economy and Competitiveness under Grant TIN2015-66972-C5-4-R, co-financed by FEDER funds.(MINECO/FEDER/UE)

A. Jimeno-Morenilla
Department of Computer Technology, University of Alicante, E-03071, Alicante, Spain.
Tel.: +34-965903400 x 2453
Fax: +34-965909874
E-mail: jimeno@dtic.ua.es

J.L. Sánchez-Romero
Department of Computer Technology, University of Alicante, E-03071, Alicante, Spain.

H. Migallón
Department of Physics and Computer Architecture. Miguel Hernández University, E-03202 Elche, Spain.

H. Mora-Mora
Department of Computer Technology, University of Alicante, E-03071, Alicante, Spain.

1 Introduction

Optimization methods are devoted to find an optimal value for a given function, generally a minimum. Each function to be optimized has its specific domain, behaviour and number of variables involved. Indeed, some of these functions have local minima, so the find of the absolute optimum can become very difficult.

Optimization methods can be mainly divided into deterministic and heuristic approaches. Deterministic approaches take advantage of the analytical properties of the function (see [1]). When coping with non-convex or large-scale optimization problems, determining the global optimum may become a very complex task. In this case, heuristic methods should be used since they are usually more flexible and efficient than deterministic ones, and the computational time required to find the optimum can be highly reduced.

Heuristic optimization methods are classified into Evolutionary Algorithms (EA) and Swarm Intelligence (SI) algorithms. Among the EA methods, it is worthwhile mentioning Genetic Algorithm (GA), Differential Evolution (DE), Evolutionary Strategy, and some others. Among the SI methods (see [2]), it is worthwhile mentioning Particle Swarm Optimization (PSO), Artificial Bee Colony (ABC), and some others. Other methods based on phenomena in nature have been developed, such as Harmony Search, Biogeography-Based Optimization, and some others (see [3,4]). The success of most of the mentioned algorithms is greatly conditioned by their specific parameters. The proper tuning of these parameters is a crucial factor for an efficient find of the global optimum. Recently, two optimization methods have been proposed, namely TLBO (Teacher-Learner Based Optimization) [5] and Jaya [6,7]. Both optimizations algorithms have the advantage of not needing specific parameter tuning. They only require general parameters such as the number of iterations and the population dimension. Although they are very similar, TLBO uses two phases per iteration, whereas Jaya only performs one phase per iteration. The Jaya algorithm has generated a growing interest in many scientific and engineering areas due to its simplicity and efficiency, see for example [8–12] and some others.

Optimization algorithms have been usually implemented on computer systems following a traditional and sequential approach. However, most of these algorithms are feasible to be decomposed into independent tasks and executed in parallel. In the last years, the performance of parallel hardware architectures has greatly increased, while their cost has been highly reduced. Nevertheless, parallelizing an algorithm is not a simple task since it requires a reformulation and adequacy to the specific architecture to be used. Our work is based on the use of manycore platforms (Graphics Processing Units (GPU)). GPUs are originally dedicated to graphics processing but, since they have become massively parallel resources, they are suitable to be applied to other high performance processing tasks. Some research works can be found in the literature which demonstrate the advantages of executing parallel implementations of op-

timization algorithms on both multiprocessors (see for example [13–15]) and GPUs (see for example [16–21]).

The rest of the paper is organized as follow: in Section 2, we present the recent Jaya optimization algorithm and its advantages; in Section 3, we will describe the GPU based parallel algorithms developed; in Section 4, we analyze the latter both in terms of parallel performance and optimization behavior; and in Section 5 some conclusions are drawn.

2 The Jaya algorithm

As mentioned earlier, the Jaya algorithm has the advantage of not requiring specific tuning parameter: ~~only~~ population size (number of different individuals) and generations (number of iterations) should be configured. This algorithm is based on the fact that the optimal solution for a given problem can be obtained moving towards the best partial solution and, at the same time, avoiding the worst solution. Compared with other optimization methods such as GA, ABC, DE, PSO, and TLBO, Jaya obtained better results in terms of best, mean, and worst values of different constrained and unconstrained benchmark functions [22].

The description of the Jaya algorithm is as follows. Let $f(x)$ be the objective function to be minimized (or maximized), where x is a vector of dimension n, which depends on the particular function to be optimized. Each element of vector x is a design variable of function $f(x)$. At any iteration i there are n design variables (i.e. $j = 1, 2, \ldots n$) corresponding to the function considered, ~~assume that there are n design variables (i.e. $j = 1, 2, \ldots n$)~~ and p candidate solutions (i.e. population size, $k = 1, 2, \ldots p$), therefore the whole population can be considered as a matrix of dimension (p, n). The best candidate obtains the best value of $f(x)$ (i.e. $f(x)_{best}$) in the whole candidate solutions, and the worst candidate obtains the worst value of $f(x)$ (i.e. $f(x)_{worst}$) in the whole candidate solutions. If $X_{j,k,i}$ is the value of the jth variable for the kth candidate during the ith iteration, then this value is modified by means of the following equation:

$$X'_{j,k,i} = X_{j,k,i} + r_{1,j,i}\left(X_{j,best,i} - |X_{j,k,i}|\right) - r_{2,j,i}\left(X_{j,worst,i} - |X_{j,k,i}|\right), \quad (1)$$

where $X_{j,best,i}$ is the value of the variable j for the best candidate, and $X_{j,worst,i}$ is the value of the variable j for the worst candidate. In Equation 1, $X'_{j,k,i}$ is the updated value of $X_{j,k,i}$, and $r_{1,j,i}$ and $r_{2,j,i}$ are two random numbers, in the range [0, 1], for the jth variable computed in the ith iteration. The term $r_{1,j,i}\left(x_{j,best,i} - |X_{j,k,i}|\right)$ indicates the tendency of the algorithm to move closer to the best solution, whereas the term $-r_{2,j,i}\left(x_{j,worst,i} - |X_{j,k,i}|\right)$ indicates the tendency of the algorithm to avoid the worst solution. The new candidate ($X'_{j,k,i}$) is accepted only if it gives a better function evaluation. All the function values accepted at the end of each iteration are maintained, so these values become the input to the next iteration.

Moreover, the Jaya algorithm performs several independent executions (Runs) of the algorithm. As mentioned earlier, Jaya is free of tuning parameters, in addition to the population size and number of generations, the third input parameter is the number of independent executions. Considering all the computed solutions (one of each independent execution), statistical data about these results (best, worst and mean solution, and also standard deviation) are the algorithm output.

Algorithm 1 shows the skeleton of the Jaya algorithm, in line 4 a new population of *PopSize* members is created, in line 6, i.e. in each iteration *PopSize* members are created following (1), each new member is compared with the member of the current population that is replaced if the new member is better. Finally, in line 8, the solution (the best member of the current population) is stored, the current population is removed (line 9) starting a new execution of the Jaya procedure. The output of the algorithm are composed by the global best solution and the statistical data about the *Runs* solutions obtained in each independent execution. Note that each execution of Jaya procedure starts with a whole new population, being completely independent from the rest of the executions, since we ensure that the seed used for the generation of random numbers will be different for each execution.

Algorithm 1 Sequential Jaya algorithm

```
1: Input parameters: Runs, Iterations, PopSize
2: Define function to minimize
3: for exec = 1 to Runs do
4:     Create New Population
5:     for iter = 1 to Iterations do
6:         Update Population
7:     end for
8:     Store Solution
9:     Delete Population
10: end for
11: Obtain Best Solution and Statistical Data
```

3 GPU acceleration of Jaya

The Jaya algorithm has inherent parallel features which can be exploited. On the one hand, each candidate solution (individual $k = 1, 2, \ldots p$ in the algorithm) into the population can independently perform the function evaluation. Moreover, each design variable ($j = 1, 2, \ldots n$ in the algorithm) can update its value, taking into account only the current best and worst values. ~~Moreover, the Jaya algorithm performs several independent executions (Runs) of the algorithm. Considering all the computed solutions, statistical data about the results (best, worst and mean solution, and also standard deviation) are the algorithm output.~~ ~~The parallel Jaya implementation was developed using~~

~~CUDA 7.5. The Nvidia GTX970 (1,664 CUDA cores, 1.025GHz, 2GB memory) Maxwell GPU was used to evaluate the parallel performance with respect to the sequential implementation, which was supported by a general purpose Intel processor (i7-6700 (3.4GHz)). The GPU used is a Maxwell Nvidia GPU, which therefore is composed by Maxwell Streaming Multiprocessors (SMM), each one with 128 CUDA cores.~~

Is worthy to note that depending on the function to be optimized, the computational cost of the Jaya algorithm may be small respect to the required synchronization processes, which can make the work of parallelization unsuccessful. For example, in a first parallel approach, in order to increase the computational cost assigned to each CUDA core, a single CUDA thread is responsible of the whole computation of the function to be minimized (or maximized), i.e. the computing of the function evaluation of one individual of population. Therefore, taking into account that the maximum size of population is usually not more than several hundreds, the number of total number of CUDA threads may be not enough to occupy efficiently the GPU. As reference, we have developed this parallel approach obtaining a low speed-up of $10x$ when comparing the GPU (parallel) and the CPU (sequential) executions for Rosenbrock function. In our proposal parallel approach we increase the number of CUDA threads to increase the occupancy of the GPU, for that purpose we analyze the 30 test functions to exploit the inherent parallelism inside them, increasing the total number of CUDA threads. Note that, the parallel computation of the evaluation of the functions involves reduction processes, which, should be performed using the GPU shared memory to compute them efficiently.

As can be seen in (1) the entire updated population is accessed by all threads involved in the computing, in order to read best and worst individuals and to update them when necessary. Therefore, in order to obtain a good parallel performance, the population data should be stored in GPU shared memory. Since the GPU shared memory is owned by each GPU multiprocessor (SMM in our GPU), each independent execution of the algorithm should be mapped on one single SMM, i.e. the number of CUDA thread blocks in the grid of the kernel launched is equal to the desired number of independent executions (Runs), and, as aforementioned, the number of CUDA threads per block depends on both the population size and low level parallelization of the function to be optimized.

In our proposal, the number of threads per block are set considering both the number of design variables of the function to be optimized and the population size. Accordingly, the number of threads per block were configured in a 2D array, being the row size equal to the population size and being the column size equal to the number of design variables. As previously mentioned, the GPU shared memory is key to obtain an efficient behavior due to is used to store the whole population, the partial values of the evaluations of the functions, the new candidates, and other data related to the implementation of the algorithm, for example, the indices of the current best and worst solution.

Algorithm 2 shows the skeleton of the parallel implementation of Jaya in the GPU platform. The low level inherent parallelism of functions to be optimized are, usually, linked to computation associated to each design variable, therefore i iterates over the population size (line 2) and j iterates over the number of design variables (line 3), while each thread block obtains a solution to be transferred to CPU (line 10). After each update population (line 6), a thread synchronization barrier is needed to start the new iteration with the correct values (best and worst) of population.

Algorithm 2 Skeleton of the parallel Jaya GPU implementation (kernel).

```
1: DEVICE CODE:
2: i = threadIdx.x i = 0, 1, ... Population
3: j = threadIdx.y j = 0, 1, ... VARs
4: Create Population (i, j)
5: for k = 1 to Iterations do
6:     Update Population (i, j)
7:     Synchronization barrier
8: end for
9: exec = blockIdx.x
10: Store Best Solution Solution(exec)
11: Delete Population
12:
13: HOST CODE:
14:   Obtain Best Solution and Statistical Data
```

The operations performed by each thread for creating the different individuals (line 4 in Algorithm 2), are depicted in Algorithm 3. Note that the independent computation associated to each design variable are denoted by $Fs(i, j, var)$ in line 5, a reduction procedure is used to obtain the function evaluation of each population member (line 8). Once the function is evaluated for all population members, in line 10 best and worst candidates are obtained after the necessary synchronization barrier.

Algorithm 3 Create Population (device function).

```
1: Create Population Function. Block Size (Pop, VAR):
2: {
3:     Obtain random number r
4:     Compute design var (var(i, j, r))
5:     Compute Fs(i, j, var): independent function term (i, j)
6:     Synchronization barrier
7:     REDUCTION process to obtain (GPU Shared Memory):
8:         F_start(i): function evaluation of member i
9:     Synchronization barrier
10:    REDUCTION process to obtain (GPU Shared Memory):
11:        Best and Worst solutions of Population
12: }
```

Finally, Algorithm 4 shows the operations performed for updating each individual (line 6 in Algorithm 2). In line 5 a new design variable is computed following (1) and we compute the function evaluation part associated to this design variable, to obtain the function evaluation after the synchronization barrier in line 9. Depending on the function evaluation the new design variable are stored or not (line 12). Analogously to Algorithm 3 best and worst candidates are obtained after the necessary synchronization barrier (line 15).

Algorithm 4 Update Population (device function).

```
1: Update Population Function. Block Size (Pop, VAR):
2: {
3: Obtain random number r1
4: Obtain random number r2
5: Compute varnew(i, j): design var (i, j)
6: Compute Fs(i, j): independent function term (i, j)
7: Synchronization barrier
8: REDUCTION process to obtain (GPU Shared Memory):
9:     F_new(i): function evaluation of member i
10: Synchronization barrier
11: if F_new(i) < F(i) then
12:     var(i, j) = varnew(i, j)
13: end if
14: REDUCTION process to obtain (GPU Shared Memory):
15:     Best and Worst solutions of Population
16: }
```

In order to clarify the kernel dimensions, Fig. 1 depicts the parallel execution scheme, where each block corresponds to an independent run. Each independent execution (Run) of the algorithm is performed by one block, the solution obtained is stored in GPU global memory and transferred to the CPU to compute, in CPU, the final statistical data. The threads of one block perform the parallel computing of the partial values of the evaluations of the function for the whole population, where P is the population size (that is, number of individual) and N is the number of design variables involved in the function to be optimized. Note that the value of P is considered a tuning parameter, while the value of N is inherent to the function to be optimized. Worthy to note that the GPU shared memory performance and the available amount is a key to obtain good speed-ups. On the one hand, the shared memory performance allows sharing efficiently the data involved in the calculations; so, the amount of shared memory allows increasing the population size.

Algorithm 5 shows the procedure used to perform reductions in the GPU, based on sequential addressing of shared memory, which is conflict free, and using a reversed loop and threadID-based indexing. Depending of the particular reduction operation to be performed *tam* (line 2 could be equal to the number of design variables (for example to perform a function evaluation) or could be equal to population size (for example to obtain best and worst members).

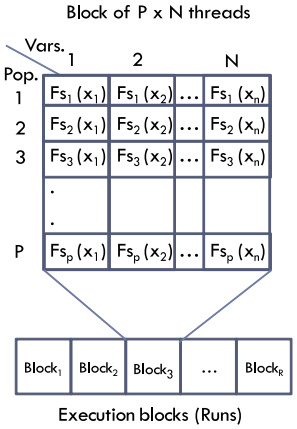

Fig. 1 Parallel computing scheme on GPU.

Algorithm 5 CUDA reduction procedures.

```
1: Obtain thread identification tid inside the thread block
2: for s = tam;s > 0;s >>= 1 do
3:    if tid < s then
4:       Perform reduction operation over the shared memory
5:    end if
6:    CUDA synchronization barrier
7: end for
```

4 Numerical experiments

The comparison between the sequential and the parallel implementations of the algorithm was made taking into account 30 unconstrained functions frequently used as a well-known benchmark in several works about optimization (see for example [7,23]), listed in Table , where VAR is the number of design variables of each function.Note that depending on the particular function to be optimized it will be achieved different speed-ups.

The parallel Jaya implementation was developed using CUDA 7.5. The Nvidia GTX970 ($1,664$ CUDA cores, $1.025GHz$, $2GB$ memory) Maxwell GPU was used to evaluate the parallel performance with respect to the sequential implementation, which was supported by a general purpose Intel processor (i7-6700 ($3.4GHz$)). The GPU used is a Maxwell Nvidia GPU, which therefore is composed by Maxwell Streaming Multiprocessors (SMM), each one with 128 CUDA cores.

In our experiments the number of iterations was set to $30,000$, in order to ensure the Jaya convergence even for the smaller population sizes, considering a maximum error of 10^{-5}. Two parameters were modified so as to evaluate the parallel algorithm when compared with the sequential implementation in terms of both the speed-up and the optimization performance: number of Runs was varied in the set of values $\{2, 4, 8, 16, 32, 64, 128, 256\}$; and population size

Table 1 Benchmark functions.

Id.	Function	VAR	Id.	Function	VAR
F1	Sphere	30	F16	Booth	2
F2	SumSquares	30	F17	Michalewicz 2	2
F3	Beale	5	F18	Michalewicz_5	5
F4	Easom	2	F19	Bohachevsky_2	2
F5	Matyas	2	F20	Bohachevsky_3	3
F6	Colville	4	F21	GoldStein-Price	2
F7	Trid_6	6	F22	Perm	4
F8	Trid_10	10	F23	Hartman_3	3
F9	Zakharov	10	F24	Ackley	30
F10	Schwefel_1.2	30	F25	Penalized_2	30
F11	Rosenbrock	32	F26	Langerman_2	2
F12	Dixon-Proce	30	F27	Langerman_5	5
F13	Foxholes	2	F28	Langerman_10	10
F14	Branin	2	F29	FletcherPowell_5	5
F15	Bohachevsky 1	2	F30	FletcherPowell_10	10

was varied within the set $\{8, 16, 32\}$. Due to the number of Runs sets the number of thread blocks, and the number of individuals joint to the number of the design variables set the number threads per block, these two parameters set the grid dimensions to launch the kernel.

Figure 2 shows the speed-up achieved when comparing the parallel implementation on the Nvidia GTX970 GPU with the sequential execution on the Intel i7-4790 processor, for the optimization of the Rosenbrock function which is defined with 30 design variables. Note that when the value of Runs is smaller than the number of SMM (13 for the Nvidia GTX970) the GPU cannot be fully occupied and, therefore, the speed-up obtained is usually low. Therefore, the number of Runs must be at least equal to the number of multiprocessors of the GPU, moreover is well known that to obtain good occupancy the number of thread blocks, usually, must be greater than the number of available multiprocessors. It can be observed that a maximum speed-up higher than $50x$ was obtained with 64 Runs and a population size of 8. In this case, the error was in the order of 10^{-3} for both the sequential and the parallel implementations. Note that the speed-up improves as the population size decreases for the Rosenbrock function. This fact is due to the reduction processes become less significant with respect to the total computational cost. The optimal population size depends on both the computational cost of the function evaluation to be optimized and the number of variables of this function. So, we can not set a global optimal value of population size.

On the other hand, in Table 2, we analyze both the error and the speed-up obtained in the parallel optimization procedure for Rosenbrock function, varying the population size and Runs but remaining unchanged the number of evaluations of the function $(7,680,256)$. For that, the results shown in this table are obtained setting the number of iterations equal to $30,000$ and the product of population size by Runs is always equal to 256. As expected, the error of the function increases for very low population size (8) while the error

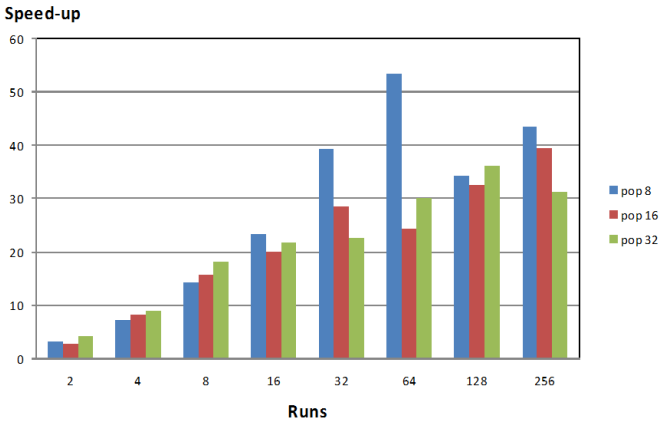

Fig. 2 Speed-up obtained for the GPU Jaya algorithm to optimize Rosenbrock function with different number of independent executions (Runs) and population sizes.

of the function is similar for populations sizes equal to 16 and 32. Note that, the convergence ratio of the algorithm improves when the number of individuals in which to look for the best and worst individual grows. On the other hand, a high increase of Runs performed, remaining a very low population size, may not improve the convergence ratio due to the Runs executions are independent executions. The speed-up increases as the number of Runs increases, as aforementioned the occupancy of the GPU depends, among other parameters, on the number of Runs, which is the number of thread blocks, therefore the number of Runs always must be greater than the number of streaming multiprocessors of the GPU. Therefore the pair of values population size and Runs must be chosen taking into account both the convergence ratio and the acceleration of the parallel algorithm.

Table 2 Error and speed-up obtained for the GPU Jaya algorithm to optimize Rosenbrock function with the same number of function evaluations, varying population size and Runs.

Rosenbrock function with $7,680,256$ function evaluations			
Runs x Population size	8 x 32	16 x 16	32 x 8
Function error	8.84E-28	2.11E-27	2.90E-05
Speed-up	18.2	20.2	39.4

In order to analyze the optimal values of Runs and population size, we have developed the CUDA parallel optimization algorithm of the whole benchmark. Speed-up was calculated by following the same criteria as with the Rosenbrock function with regard to fixed number of iterations, number of independent executions, and population size. Remark that, depending on the function to be optimized, the value of Runs were increased up to 1024, whereas in other

cases this parameter had to be decreased to 128 or even 64 due to the features of the GTX970 GPU.

Table 3 shows the value of Runs and the population size related to the maximum speed-up obtained for each one of the benchmark functions. First, we want to highlight that the performance of sequential and parallel algorithms are very close, but attending to the characteristics of the Jaya algorithm (see (1)), one execution, either sequential or parallel, can not be the exactly equal to another execution.

Table 3 Maximum speed-up for the 30 test functions.

Id.	Runs	Pop.	Max. speed-up	Id.	Runs	Pop.	Max. speed-up
F05	1024	64	189.6	F09	64	32	41.3
F29	128	32	132.0	F08	512	8	40.7
F30	256	16	114.4	F02	128	16	40.3
F22	512	32	91.3	F12	256	8	39.5
F17	256	64	73.5	F04	512	64	38.6
F26	512	64	67.8	F18	64	128	37.1
F28	128	32	54.6	F13	256	64	36.5
F11	64	8	53.5	F21	128	64	35.4
F16	256	256	52.1	F14	256	128	31.9
F03	128	64	47.3	F15	512	128	31.1
F27	256	32	46.7	F06	128	32	31.0
F07	512	32	44.6	F19	256	64	28.6
F10	32	16	42.9	F25	128	8	26.2
F01	256	8	42.8	F23	64	64	23.8
F20	512	64	41.8	F24	128	8	18.4

On the other hand, it can be observed that we obtain good speed-ups values being, in some cases, the speed-up higher than $100x$. Indeed, in case of F05 (the Matyas function), the speed-up is near $190x$ with a population size of 64 and 1024 Runs. Analyzing Table 3, we can extract some conclusions, for example functions Trid_6 and Trid1_10 have the same definition, not more complex, being the only difference the number of design variables (6 and 10 respectively), which definition is shown in (2), where the number of design variables is represented by D, being $D = 6$ for Trid_6 and $D = 10$ for Trid_10. Both functions obtain the maximun speed-up with the same number of Runs (equal to 512) but the population size must be greater for Trid_6, i.e. the function with lower number of variables. Similar conclusions can be applied to the Langermann (F26, F27 and F28) and the Fletcher-Powell (F29 and F30) functions.

$$F_{Trid} = \sum_{1}^{D} \left(x_i^2 - 1\right)^2 - \sum_{2}^{D} \left(x_i * x_{i-1}\right) \tag{2}$$

However, with regard to the Michalewicz (F17 and F18) functions, maximum speed-up significantly increases when decreasing the number of design

variables from 5 to 2, i.e. optimizing the Michalewicz function with $D = 2$ instead of with $D = 5$. In this case, the high complexity of the Michalewicz functions, shown in (3), causes not efficient computation in the GPU.

$$F_{Michalewicz} = -\sum_{1}^{D} \sin x_i \left(\sin \left(\frac{i x_i^2}{\pi} \right) \right)^{20} \tag{3}$$

The best speed-up is obtained optimizing the Matyas function, shown in (4). In this function, each thread i computes the corresponding term $0.26x_i^2$, while the first thread also performs the final summation of the two terms with the last one. In order to follow the parallel scheme shown in Fig. 1 the number of threads to compute one Matyas function evaluation is equal to 2, i.e. the number of design variables, it could be considered the computational load is unbalanced because, after computing the terms $0.26x_i^2$, only one thread works in order to compute the term $0.48x_1x_2$ and to reduce the three partial results to obtain the final result. Note that in this case only one thread store its partial term computed in the GPU shared memory to be read for the thread that computes the final result. So the percentage of idle threads while computing the functions evaluations is only 50%.

$$F_{Matyas} = 0.26(x_1^2 + x_2^2) + 0.48x_1x_2 \tag{4}$$

The worst speed-up is obtained optimizing the Ackley function, shown in (5), in which each thread i computes in parallel the terms x_i^2 and $\cos 2\pi x_i$. Indeed, some threads participate in the reduction (summation) of the aforementioned terms, and finally, the first thread performs the exponentiation calculations and the summation of the four terms to provide the global function evaluation. Adding that the number of variables is high (i.e. $D = 30$) the numerous reduction processes causes, in this case, the speed-up obtained.

$$F_{Ackley} = -20 \exp \left(-0.2 \sqrt{\frac{1}{D} \sum_{1}^{D} x_i^2} \right) - \exp \left(\frac{1}{D} \sum_{1}^{D} \cos 2\pi x_i \right) + 20 + e \tag{5}$$

A high number of variables does not imply poor performance, for example the Rosenbrock function has a large number of design variables (30) but it obtains good speed-up values of $53.5x$, even better than several functions with a smaller number of design variables. The definition of this function is shown in (6), in which $D = 30$. In this function, each thread i, except the last one, computes the term $100(x_i^2 - x_{i+1})^2 + (1 - x_i)^2$. Once all terms have been computed only remains to perform the reduction (summation) procedure, in which some threads participate.

$$F_{Rosenbrock} = \sum_{1}^{D-1} \left(100 \left(x_i^2 - x_{i+1} \right)^2 + \left(1 - x_i \right)^2 \right) \tag{6}$$

5 Conclusions

In this work we have presented a GPU based parallel algorithm of Jaya, a recent optimization algorithm. We have described in detail the different levels of the parallel algorithms developed. We have analyzed the performance of our proposals using a benchmark of 30 unconstrained functions, which have not been specifically optimized, although in some of them it is advisable. Although good performance has been achieved for the whole benchmark, it has been shown in what kind of functions the performance could be improved. We conclude, as the results of the experimentation demonstrate, that the parallel implementation of Jaya on GPUs provides a significant speed-up when compared with the sequential execution in CPUs. In the best case, the speed-up rises to near $190x$, being the mean speed-up for the 30 benchmark functions equal to 53x, whereas the median is $41x$.

References

1. M. H. Lin, J. F. Tsai, and C. S. Yu, "A review of deterministic optimization methods in engineering and management," *Mathematical Problems in Engineering*, vol. 2012, 2012.
2. E. Bonabeau, M. Dorigo, and G. Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, 1999.
3. R. V. Rao and V. Patel, "An elitist teaching-learning-based optimization algorithm for solving complex constrained optimization problems," *International Journal of Industrial Engineering Computations*, vol. 3, pp. 535–560, 2012.
4. ——, "Comparative performance of an elitist teaching-learning-based optimization algorithm for solving unconstrained optimization problems," *International Journal of Industrial Engineering Computations*, vol. 4, pp. 29–50, 2013.
5. R. V. Rao, V. Savsani, and D. Vakharia, "Teaching-learning-based optimization: A novel method for constrained mechanical design optimization problems," *Computer-Aided Design*, vol. 43, no. 3, pp. 303–315, 2011.
6. R. V. Rao, D. P. Rai, and J. Balic, "A multi-objective algorithm for optimization of modern machining processes," *Engineering Applications of Artificial Intelligence*, vol. 61, no. Supplement C, pp. 103 – 125, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0952197617300465
7. R. V. Rao, "Jaya: A simple and new optimization algorithm for solving constrained and unconstrained optimization problems," *International Journal of Industrial Engineering Computations*, vol. 7, pp. 19–34, 2016.
8. S. P. Singh, T. Prakash, V. Singh, and M. G. Babu, "Analytic hierarchy process based automatic generation control of multi-area interconnected power system using Jaya algorithm," *Engineering Applications of Artificial Intelligence*, vol. 60, pp. 35–44, 2017.
9. K. Gao, Y. Zhang, A. Sadollah, and R. Su, "Jaya algorithm for solving urban traffic signal control problem," in *Control, Automation, Robotics and Vision (ICARCV), 2016 14th International Conference on*. IEEE, 2016, pp. 1–6.
10. R. Azizipanah-Abarghooee, M. Malekpour, M. Zare, and V. Terzija, "A new inertia emulator and fuzzy-based lfc to support inertial and governor responses using Jaya algorithm," in *Power and Energy Society General Meeting (PESGM), 2016*. IEEE, 2016, pp. 1–5.
11. M. Bhoye, M. Pandya, S. Valvi, I. N. Trivedi, P. Jangir, and S. A. Parmar, "An emission constraint economic load dispatch problem solution with microgrid using Jaya algorithm," in *Energy Efficient Technologies for Sustainability (ICEETS), 2016 International Conference on*. IEEE, 2016, pp. 497–502.

12. I. N. Trivedi, S. N. Purohit, P. Jangir, and M. T. Bhoye, "Environment dispatch of distributed energy resources in a microgrid using Jaya algorithm," in *Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB), 2016 2nd International Conference on*. IEEE, 2016, pp. 224–228.
13. A. J. Umbarkar, M. S. Joshi, and P. D. Sheth, "Openmp dual population genetic algorithm for solving constrained optimization problems," *International Journal of Information Engineering and Electronic Business*, vol. 1, pp. 59–65, 2015.
14. R. Baños, J. Ortega, and C. Gil, "Comparing multicore implementations of evolutionary meta-heuristics for transportation problems," *Annals of Multicore and GPU Programming*, vol. 1, no. 1, pp. 9–17, 2014.
15. P. Delisle, M. Krajecki, M. Gravel, and C. Gagné, "Parallel implementation of an ant colony optimization metaheuristic with openmp," in *Proceedings of the 3rd European Workshop on OpenMP*. Springer Berlin Heidelberg, 2001.
16. Y. Tan and K. Ding, "A survey on gpu-based implementation of swarm intelligence algorithms," *IEEE Transactions on Cybernetics*, vol. 46, no. 9, pp. 2028–2041, Sept 2016.
17. G. H. Luo, S. K. Huang, Y. S. Chang, and S. M. Yuan, "A parallel bees algorithm implementation on gpu," *Journal of Systems Architecture*, vol. 60, no. 3, pp. 271 – 279, 2014, real-Time Embedded Software for Multi-Core Platforms. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762113001872
18. A. Delvacq, P. Delisle, M. Gravel, and M. Krajecki, "Parallel ant colony optimization on graphics processing units," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 52 – 61, 2013, metaheuristics on GPUs. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731512000044
19. L. Mussi, F. Daolio, and S. Cagnoni, "Evaluation of parallel particle swarm optimization algorithms within the cuda architecture," *Information Sciences*, vol. 181, no. 20, pp. 4642 – 4657, 2011, special Issue on Interpretable Fuzzy Systems. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025510004263
20. L. de P. Veronese and R. A. Krohling, "Differential evolution algorithm on the gpu with c-cuda," in *IEEE Congress on Evolutionary Computation*, July 2010, pp. 1–7.
21. Y. Zhou and Y. Tan, "Gpu-based parallel particle swarm optimization," in *2009 IEEE Congress on Evolutionary Computation*, May 2009, pp. 1493–1500.
22. R. V. Rao and G. Waghmare, "A new optimization algorithm for solving complex constrained design optimization problems," *Engineering Optimization*, vol. 49, no. 1, pp. 60–83, 2017.
23. D. Karaboga and B. Akay, "A comparative study of artificial bee colony algorithm," *Applied Mathematics and Computation*, vol. 214, no. 1, pp. 108 – 132, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0096300309002860