IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 5, 2019, accepted August 22, 2019, date of publication September 12, 2019,
date of current version September 27, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2941086

Comparison of High Performance Parallel Implementations of TLBO and Jaya Optimization Methods on Manycore GPU

H. RICO-GARCIA[1], JOSE-LUIS SANCHEZ-ROMERO[1], A. JIMENO-MORENILLA[1], H. MIGALLON-GOMIS[2], H. MORA-MORA[1], AND R. V. RAO[3]

[1]Department of Computer Technology, University of Alicante, 03690 Alicante, Spain
[2]Department of Computer Engineering, Miguel Hernández University, 03202 Elche, Spain
[3]Sardar Vallabhbhai National Institute of Technology, Surat 395 007, India

Corresponding author: Jose-Luis Sanchez-Romero (sanchez@dtic.ua.es)

This work was supported in part by the Spanish Ministry of Economy and Competitiveness under Grant TIN2017-89266-R, in part by FEDER funds (MINECO/FEDER/UE), and in part by the Spanish Ministry of Science, Innovation, and Universities co-financed by FEDER funds under Grant RTI2018-098156-B-C54.

ABSTRACT The utilization of optimization algorithms within engineering problems has had a major rise in recent years, which has led to the proliferation of a large number of new algorithms to solve optimization problems. In addition, the emergence of new parallelization techniques applicable to these algorithms to improve their convergence time has made it a subject of study by many authors. Recently, two optimization algorithms have been developed: Teaching-Learning Based Optimization and Jaya. One of the main advantages of both algorithms over other optimization methods is that the former do not need to adjust specific parameters for the particular problem to which they are applied. In this paper, the parallel implementations of Teaching-Learning Based Optimization and Jaya are compared. The parallelization of both algorithms is performed using manycore GPU techniques. Different scenarios will be created involving functions frequently applied to the evaluation of optimization algorithms. Results will make it possible to compare both parallel algorithms with regard to the number of iterations and the time needed to perform them so as to obtain a predefined error level. The GPU resources occupation in each case will also be analyzed.

INDEX TERMS CUDA, GPU, Jaya, TLBO, optimization, parallelism.

I. INTRODUCTION

The resolution of engineering problems using advanced computer techniques is a field in continuous evolution, among which stands out the problems of optimization in the search for solutions. The characteristics of this type of problems are the following: there are a series of objectives, a set of possible solutions (with all the possible values generated by the design parameters) and a series of processes for finding a solution to the problem (optimization methods). The optimization methods look for the optimal solution within the set of feasible solutions. Continuous research is underway in this field and heuristic optimization methods inspired by nature are proving to be better in many cases than deterministic methods and are therefore becoming more frequently used.

There are many optimization algorithms inspired by nature that use principles observed in different natural phenomena. Each of these algorithms is based on a specific phenomenon observed within nature. For example, Genetic Algorithm (GA) uses the theory of evolution to improve the population towards the solution of the problem; Particle Swarm Optimization (PSO) emulates the behaviour of bird swarms in search of food; the Artificial Bee Colony (ABC) and the Ant Colony Optimization (ACO) algorithms are inspired by the organizations of the colonies of both types of insects. Due to the impressive advances in recent years in the field of parallel architectures, while the cost of the same has been greatly reduced, most of these optimization algorithms have been parallelized so as to obtain an increase

The associate editor coordinating the review of this manuscript and approving it for publication was Shaoyong Zheng.

in performance [1]–[10]. The application of these algorithms to the solution of optimization problems within engineering is proven in many works with satisfactory results.

One of the disadvantages of these algorithms generally is the use of a large number of parameters and restrictions specific to each of the algorithms. These parameters and restrictions usually vary according to the problem to be addressed, and also have a direct influence on the outcome of the application of the algorithm and its effectiveness. In order to deal with these problems, new optimization algorithms based on population evolution have been proposed, but avoiding the use of parameters and restrictions. Two recent optimization algorithms follow this approach: Teaching-Learning Based Optimization (TLBO) and Jaya. These two algorithms do not use specific parameters or restrictions, but only use the parameters intrinsic to these types of algorithms such as the size of the population, the number of variables of each individual depending on the problem to be optimized, and the number of iterations. TLBO and Jaya have proven to be more efficient than other optimization algorithms when applied to the search for the optimal value of different functions of diverse complexity [11], [12], [16]. For this reason, research related to both algorithms has reached a considerable rise in recent years, and they continue to be studied, improved and applied in a wide variety of fields.

In this work, TLBO and Jaya have been selected so as to analyze and compare the performance obtained when running parallel implementations of both algorithms supported by a manycore GPU (Graphics Processing Unit) architecture. The manycore GPU platform used in this work is NVidia CUDA (Computer Unified Device Architecture). The main aim of the comparison developed is devoted to deduce which of the two parallel algorithms is more efficient and whether the complexity of the problems to be optimized (basically related to the number of variables involved and the type of operations carried out) influence the performance of each of them.

This paper is structured as follows: in Section II, the features of TLBO and Jaya will be shown; in Section III, the parallel approaches of both algorithms based on the manycore GPU CUDA architecture are explained; Section IV shows the features and result of the experiments performed; finally, in Section V, conclusions of the research are summarized and future works are proposed.

II. TLBO AND JAYA FEATURES

As mentioned before, TLBO and Jaya optimizations algorithms have the advantage of not needing specific parameter tuning [11]–[25]. They only require general parameters such as number of iterations and population dimension. Although they are very similar, TLBO uses two stages each one of iterations (Teacher and Learning stages), whereas Jaya only performs one stage each one of the iterations. The Jaya algorithm has generated a growing interest in many scientific and engineering areas due to its simplicity and efficiency. The TLBO algorithm usually converges to the solution faster than Jaya, but TLBO is more complex than Jaya, not only for having two stages, but also due to the operations needed in each stage which are more time-consuming than the ones of Jaya. In this research work, the original version of the Jaya and TLBO algorithms will be used, although some new versions can be found in the literature [26], [27].

A. THE TLBO ALGORITHM

TLBO is an efficient optimization method which has been used for engineering problems among others [13], [28]–[30]. This method looks for a teacher (best individual) that will probably cause an influence on the learners (the rest of individuals) to improve their features. As a population-based method, it uses a population of individuals (candidate solutions) to infer to the global solution. The TLBO algorithm is divided into two phases: the first phase is the *Teacher Phase* and the second one is the *Learner Phase*. In the *Teacher Phase*, the individuals learn from the teacher (the best solution of the whole population), and in the *Learner Phase* the individuals try to learn by means of the interaction from other individuals in the population.

The behaviour of the TLBO algorithm is as follows. The population is created and the initialization of the individuals (values of the design variables) is made with random data. After creating the first generation of individuals, the algorithm will iterate updating the population. At the start of each one of the iterations, the population *Teacher* (best solution) is determined and the mean of each design variable is calculated. These data is used in the two main stages of the algorithm: the *Teacher Stage* and the *Learner Stage*. In the *Teacher Stage*, the obtained Teacher (*Xbest*) of the current generation is used to create a new version of each individual (*Xnew*) using the following equation:

$$X_{new}(i,j) = X(i,j) + rand(0,1)\left(X_{best}(j) - TFactor \cdot X_m(j)\right) \tag{1}$$

In (1), $X(i,j)$ corresponds to the design variable j of the individual i, and it is modified by using the value of the Teacher $X_{best}(j)$, the variable mean $X_m(j)$, and the *TFactor*. *TFactor* can adopt the integer value 1 or 2 and is calculated using the following expression:

$$TFactor = round(1 + rand(0,1)) \tag{2}$$

After generating a new individual, it is submitted for evaluation. If the evaluation result is better than that of the original individual, this old individual is replaced by the new one.

In the *Learner Stage*, each individual is assigned a random contestant in the population. Both individuals are submitted for evaluation. The individual with a better evaluation is labelled as the *Partial Teacher*, and the other one is labelled as the *Learner*, so that they are used to generate a new individual using the following expression:

$$X_{new}(i,j) = X(i,j) + rand(0,1) \cdot (PartialTeacher(j) - Learner(j)) \tag{3}$$

When every $Xnew(i, j)$ is generated, the new individual is evaluated and compared with the original individual. If the

evaluation of the new individual improves that of the old one, the new individual replaces the original one.

B. THE JAYA ALGORITHM

The Jaya algorithm is a populated-based optimization method so as to calculate optimal solutions for constrained and unconstrained optimization problems. Unlike other population-based heuristic algorithms, Jaya has no algorithm-specific controlling parameter or tuning parameters. As in TLBO, only population size 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 shifting towards the best partial solution and, at the same time, evading the worst solution. Compared with other optimization methods, Jaya obtained better results in terms of best, mean, and worst values of different unconstrained benchmark functions [16]. Similarly to TLBO, Jaya has been recently applied to optimizing a wide variety of engineering problems [31]–[38].

The description of the Jaya algorithm is as follows. Let $f(x)$ be the objective function to be minimized (or maximized). At any iteration i, assume that there are n design variables (i.e. $j = 1, 2, \ldots, n$) and p candidate solutions (i.e. population size, $k = 1, 2, \ldots, p$). 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:

$$X'_{j,k,i} = X_{j,k,i} + r_{1,j,i}\left(X_{j,best,i} - \left|X_{j,k,i}\right|\right) - r_{2,j,i}\left(X_{j,worst,i} - \left|X_{j,k,i}\right|\right) \quad (4)$$

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 (4), $X'_{j,k,i}$ 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 $r_{1,j,i}\left(X_{j,best,i} - \left|X_{j,k,i}\right|\right)$ indicates the tendency of the algorithm to move closer to the best solution, whereas the term $-r_{2,j,i}(X_{j,worst,i} - \left|X_{j,k,i}\right|)$ 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 accepted function values at the end of each one of the iterations are kept, so these values become the input to the next iteration.

III. PARALLEL IMPLEMENTATION

For the parallel implementation of the algorithms on the CUDA platform, a solution based on a single kernel has been used, unlike other solutions from previous works that divide the different phases and operations of TLBO into several kernels [39]. The decision to use a single kernel has been made with the aim of trying to minimize memory transfers from the CPU to the GPU and their subsequent transfer by

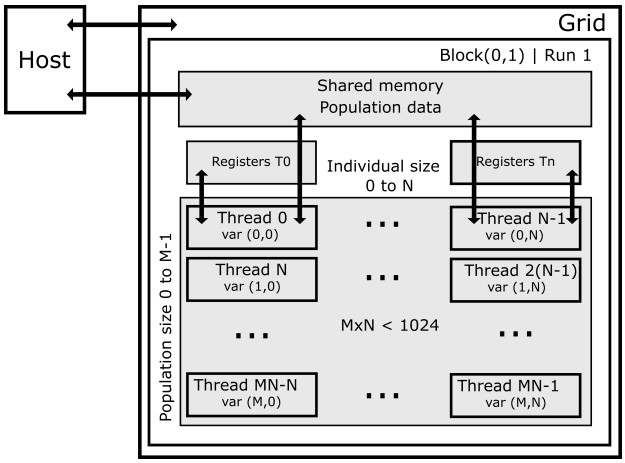

FIGURE 1. Parallel computing scheme and configuration of each block.

the different levels of memory from the GPU to the thread. The solution proposed in this work for the implementation is based on the specific CUDA architecture and an attempt has been made to establish a relationship between the different levels of organization of the algorithm data and the CUDA architecture as shown in Fig. 1.

When approaching the implementation of an algorithm through the use of parallelization techniques on GPUs, and in this case on the CUDA architecture, there are several factors that must be taken into account for a correct implementation. These types of architectures are not general-purpose, so if the algorithm to be implemented does not meet the requirements and restrictions (some of them at the programming language level) of these architectures, there may be some cases in which the algorithm cannot be implemented correctly or, if implemented, even worsen the parallel execution of the algorithm instead of improving performance. Another critical factor to take into account is memory, both in data transfer from the CPU to the GPU and its organization to improve memory access times. With regard to the organization of the memory carried out in this work, an approach has been made to enhance memory accesses by the different threads as one of the key points for improving the efficiency of CUDA implementation.

For the parallel implementation of TLBO and Jaya, the full set of GPU cores was divided into blocks. Each block is independent from the others and performs an independent run of the algorithm. At the same time, cores into a block were configured in a 2D array, so that each thread is executed on a single core. Threads within the same block share data through a shared memory bank, so each row within a block corresponds to an individual (candidate solution), whereas each column within a row corresponds to a design variable. In this way, each thread performs a partial evaluation of the individual, so a reduction operation is required for obtaining the overall evaluation (usually the addition of the partial evaluations from each thread), which is carried out by the first thread within an individual's row.

Algorithm 1 Skeleton of the Parallel TLBO GPU Implementation

```
1:for Run = 1 to Runs in parallel do //GPU block B
2: for all threads(i,j)∈(Pop,VARS) in parallel do
3:   CreatePopulation(i,j) //Create new Population
4:   EvaluatePopulation(i,j) //Evaluate function F
5:   SyncThreads
6:   for iter = 1 to Iterations do
7:     teacher = GetBest(Pop) //Get best solution
8:     SyncThreads()
9:     CalculateMean(j) //Compute mean of
       each j ∈ VARS
10:     SyncThreads()
11:     TeacherStage(i,j,teacher)
12:     SyncThreads()
13:     LearnerStage(i,j)
14:     SyncThreads()
15:   end for
16:   Store(GetBest(Pop))
17: end for
19:end for //Sequential host code:
20:Obtain Best Solution and Statistical Data
```

Algorithm 2 Teacher Stage Function for Thread(I, J) Using Teacher

```
1: TeacherStage(i,j,teacher) ∈ (Pop,VARS,Pop)
2: { 3:  r = ObtainRandomNumber()
4:  t_factor = ObtainTFactor()
5:  varnew (i,j)=
   GenerateNewIndividual(r,t_factor,
teacher)
6:  Fnew(i) = EvaluateNewIndividual
(varnew)
7:  SyncThreads
8:  if Fnew(i)<F(i) then var(i,j) = varnew(i,j)
9: }
```

Algorithm 3 Learner Stage Function for Thread(i,j)

```
1: LearnerStage(i,j) ∈ (Pop,VARS)
2: {
3:  learner = ObtainRandomLearnerFromPopulation()
4:  (r1,r2) = ObtainRandomNumbers()
5:  (best,worst) = CompareLearnerWithIndividual()
6:  varnew(i,j) =
   GenerateNewIndividual(best,r1,worst,r2)
7:  Fnew(i) = EvaluateNewIndividual(varnew)
8:  SyncThreads
9:  if Fnew(i)<F(i) then var(i,j) = varnew(i,j)
10:}
```

Algorithm 4 Skeleton of the Parallel Jaya GPU Implementation

```
1:for Run = 1 to Runs in parallel do //GPU block B
2: for all threads(i,j)∈(Pop,VARS) in parallel do
3:  CreatePopulation(i,j) //Create new Population
4:  EvaluatePopulation(i,j) //Evaluate function F
5:  SyncThreads
6:  for iter = 1 to Iterations do
7:   best = GetBest(Pop) //Gets best solution
8:   worst = GetWorst(Pop) //Gets worst solution
9:   SyncThreads()
10:   (r1,r2) = ObtainRandomNumbers()
11:   varnew(i,j) =
   GenerateNewIndividual(i,best,r1,worst,r2)
12:   SyncThreads()
13:   Fnew(i) = EvaluateNewIndividual(varnew)
14:   SyncThreads()
15:   if Fnew(i)<F(i) then var(i,j) = varnew(i,j)
16: end for
17: Store(GetBest(Pop))
18: end for
19:end for
20://Sequential host code:
21:Obtain Best Solution and Statistical Data
```

The parallel implementation of TLBO and the different phases which are carried out in each of the iteration are explained in Algorithm 1, Algorithm 2 (Teacher stage) and Algorithm 3 (Learner stage). The parallel implementation of Jaya is shown in Algorithm 4. As it can be observed, Jaya is much simpler than TLBO.

One of the problems to be solved in the implementation has been the memory restriction that CUDA has within the different levels of granularity (shared memory at block level and registers at thread level), which has led to an organization of the variables used at different levels to ensure that the complete execution of the algorithm does not need to perform memory transfers from the GPU to host or between the different levels of memory within the GPU. Another noteworthy point that has been taken into account in the implementation of the algorithm has been the blocking of the threads to perform operations (by using the specific function "*syncthreads*"). In the current literature, it is common to find references to this fact as a critical issue in CUDA implementations [40], [41], as it stops the parallel execution of threads, forcing a stop in the threads until all the threads complete the execution to that extent. In the parallel implementations, this instruction is used for all those situations where individuals must be compared or reduction calculations or function evaluation must be performed. These stops of the threads make the efficiency of the GPU to be hampered, so its use should be avoided as much as possible. Due to the nature of the algorithm that is being implemented, the synchronization order must be used in different points for a correct functioning of the parallel implementation; in addition, if the stops are not done correctly, this can lead to the use of incorrect data in the calculations to be performed. In Fig. 2 and Fig. 3, diagrams representing the implementation of each algorithm in CUDA are shown. In these diagrams, it can be observed the use of the blockages at thread level and the parallel reduction methods mentioned above.

As it can be seen in Fig. 2 and Fig. 3, the complexity of TLBO is much greater than that of Jaya, with a greater number of blockages, reductions and calculations necessary for its different phases. The extra computational complexity of TLBO achieves on the one hand the power to converge in a smaller number of iterations towards a valid solution; on the other hand, the use of the Teacher and Learner phases tries to avoid one of the big problems that this type of algorithms have: to converge on a local minimum. The use of the Learner phase aims to try to avoid this problem, using not only the reference of the best for the mutation of individuals (Teacher) but also to use other individuals among the population so as to generate mutations and evaluate them. With all this, the different nature of the functions to be minimized and

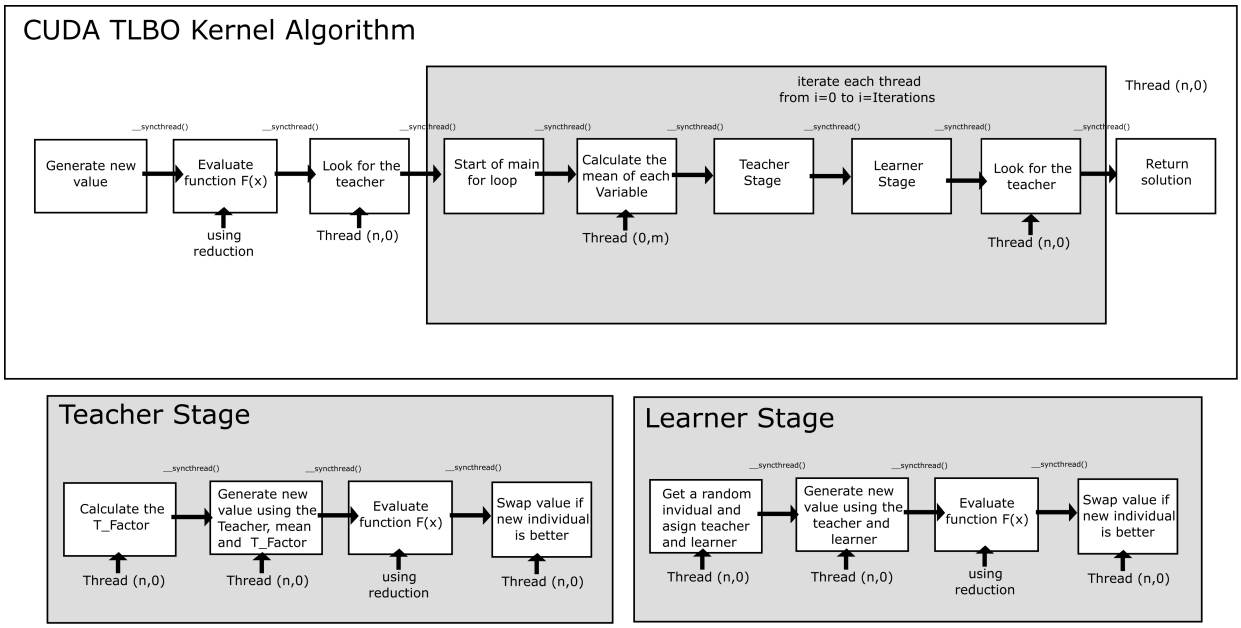

FIGURE 2. Representation of the Parallel TLBO CUDA implementation blocks and the two stages of the algorithm.

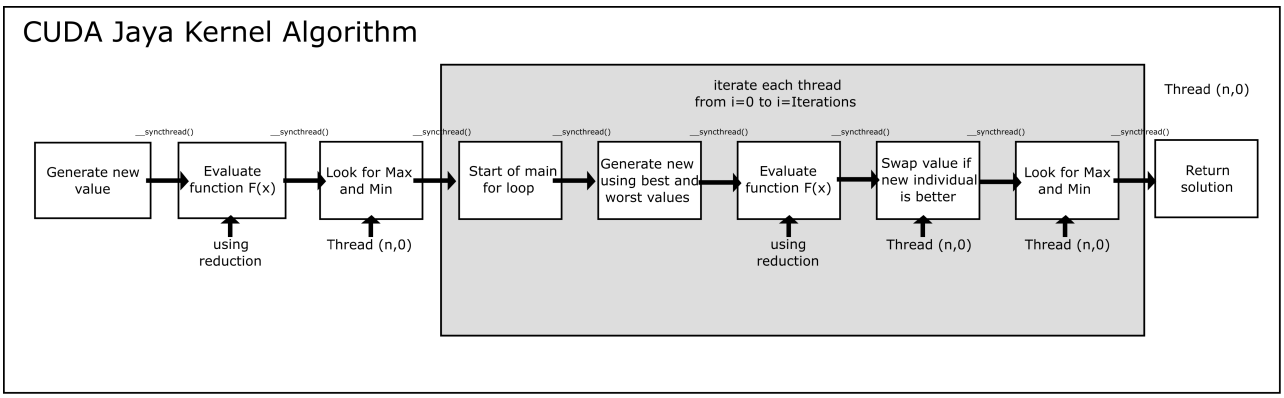

FIGURE 3. Representation of the Parallel Jaya CUDA implementation blocks and the *syncthreads*() needed.

the complexity of their calculation will take on an important weight when it comes to influencing the execution times of both algorithms. In some cases, optimization with Jaya may be favored, and although Jaya may need more iterations to be able to converge to the solution, it is possible that it will do so in a shorter time than TLBO.

IV. EXPERIMENTATION

The comparison of the CUDA parallel implementations of TLBO and Jaya was carried out using five unconstrained functions which are part of a well-known benchmark in several works about optimization [15]. The formal definition of each one of these functions is shown in Table 1. For each function, there are several parameters to tune within the different scenarios, as shown in Table 2. Some of these parameters are set following other works [6], whereas others are set after analysing the experimentation results and deducing the way in which those parameters affect the main characteristics needed for this study, like the number of iterations to find a valid solution and the execution time spent to find it.

For performing the experimentation, each function has been assigned the number of variables recommended in different research papers, and the population size has been modified for a better comparison of the algorithms. In this experiment, an admissible error level has been defined for each function so as to consider a result as a feasible solution and, therefore, to finish the execution. As seen before, Jaya is simpler than TLBO. This fact implies that Jaya parallelization is also simpler to implement in CUDA than TLBO. The main aim of the experiments consists in determining if the higher simplicity of Jaya is enough for it to achieve a greater speed when obtaining an admissible solution or, on the contrary, there are other features of both algorithms that have a greater influence on this performance index.

TABLE 1. Definition of the evaluated functions.

Fun. id	Definition
F1	$Fmin = \sum_{1}^{D} {x_i}^2$
F2	$Fmin = \sum_{1}^{D} i{x_i}^2$
F4	$Fmin = -\cos(x_1)\cos(x_2)\, e^{-(x_1-\pi)^2-(x_2-\pi)^2}$
F5	$Fmin = 0.26(x_1^2 + x_2^2) + 0.48x_1x_2$
F9	$Fmin = \sum_{1}^{D} {x_i}^2 + \left(\sum_{1}^{D} 0.5ix_i\right)^2 + \left(\sum_{1}^{D} 0.5ix_i\right)^4$
F11	$Fmin = \sum_{1}^{D-1}(100(x_i^2 - x_{i+1})^2 + (1 - x_i)^2)$

TABLE 2. Characteristics of the evaluated functions.

Function id	Function name	Number of variables	Admissible error
F1	Sphere	16	1E-10
F2	Sum of Squares	16	1E-10
F4	Easom	2	1E-250
F5	Matyas	2	1E-250
F9	Zakharov	10	1E-200
F11	Rosenbrock	16	2.5E01

The hardware used for the experimentation integrates an Intel i7 CPU at 3.4 GHz with 16GB of RAM. The GPU used for the parallel execution integrates an NVidia GeForce GTX 1060Ti with 4GB of RAM. This GPU has a Pascal architecture and the host runs the 9.1 CUDA version. The only parameter required was the population size, that is, the number of individuals, which took the values 8, 16, and 32. Each algorithm was run 32 times with the different population sizes so as to obtain average performance results. The execution of the algorithms was aimed at optimizing each of the six functions, that is, at obtaining an accurate approximation of each function's minimal.

It is worthwhile mentioning that in [15] it is demonstrated that the parallel, CUDA-based implementation of Jaya achieves higher performance than the sequential implementation. Therefore, comparing both implementations again would be redundant, so just the parallel implementations of Jaya and TLBO have been submitted for comparison.

To make the comparison of the manycore GPU implementations of the algorithms, the following results will be taken into account (all of them calculating the average values of 32 runs for each algorithm):

- Iter: Average number of iterations needed to find a valid solution of the function
- Time: Average execution time of the algorithm (seconds).
- Eval: Total number of evaluations of the function when executing the algorithm.
- Iter time: Average execution time by iteration of the algorithm.

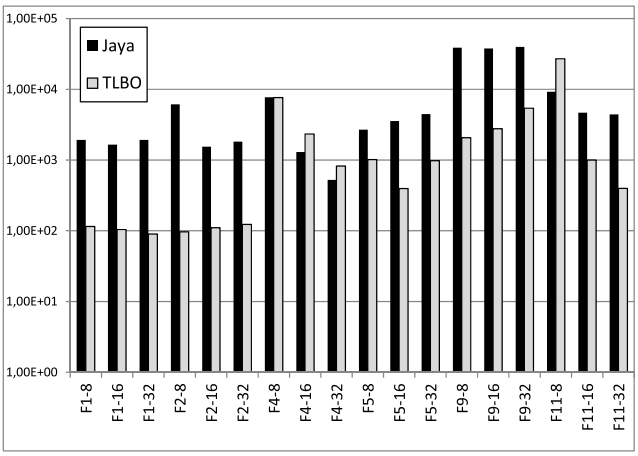

FIGURE 4. Iterations performed by Jaya and TLBO so as to achieve the required precision. Logarithmic scale.

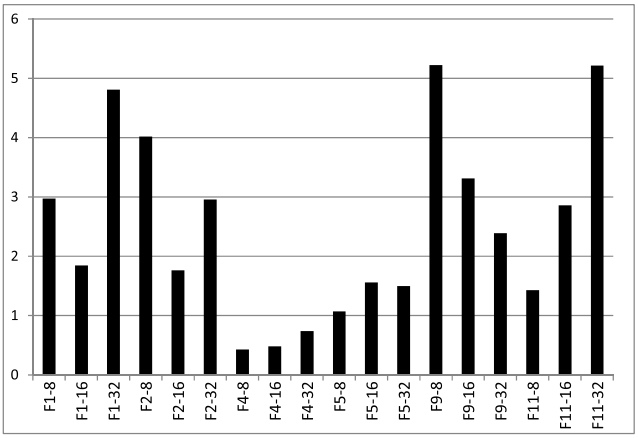

FIGURE 5. Speedup obtained by TLBO with regard to Jaya for the different functions optimized.

Results are shown in Table 3 and graphically depicted in Fig. 4 and Fig. 5. In Table 3, the shortest running time (seconds) for each of the functions and for each algorithm is shown in boldface. Moreover, Fig. 4 shows the number of iterations performed by both Jaya and TLBO to achieve the required precision for each function evaluated with different population sizes (8, 16 or 32 individuals). In this graph, a logarithmic scale is used for the Y axis (number of iterations), while the X axis indicates the functions evaluated with different population sizes (*Fi-s*, where *Fi* is the function

TABLE 3. Performance indexes for parallel jaya and TLBO.

F	Pop	Jaya				TLBO			
		iter	time	eval	iter time	iter	time	eval	iter time
F1	8	1933	5.05E-02	1.62E+06	2.61E-05	115	1.70E-02	9.03E+01	1.48E-04
F1	16	1649	**2.31E-02**	7.65E+05	1.40E-05	104	1.25E-02	1.05E+02	1.19E-04
F1	32	1933	5.05E-02	1.62E+06	2.61E-05	90	**1.05E-02**	1.15E+02	1.16E-04
F2	8	6169	4.34E-02	1.53E+06	7.04E-06	97	**1.08E-02**	1.23E+02	1.11E-04
F2	16	1548	**2.24E-02**	7.29E+05	1.44E-05	111	1.27E-02	1.11E+02	1.14E-04
F2	32	1826	5.09E-02	1.91E+06	2.78E-05	123	1.72E-02	9.72E+01	1.40E-04
F4	8	7710	7.29E-02	5.93E+05	9.46E-06	7671	1.70E-01	8.23E+02	2.22E-05
F4	16	1302	7.68E-02	2.11E+05	5.89E-05	2341	**1.59E-01**	2.34E+03	6.80E-05
F4	32	524	**1.31E-02**	7.89E+05	2.49E-05	823	1.77E-02	7.67E+03	2.15E-05
F5	8	2684	**1.52E-02**	4.45E+06	5.67E-06	1018	1.42E-02	9.85E+02	1.39E-05
F5	16	3553	1.81E-02	1.78E+06	5.10E-06	396	**1.16E-02**	3.97E+02	2.93E-05
F5	32	4457	2.25E-02	6.76E+05	5.06E-06	985	1.50E-02	1.02E+03	1.52E-05
F9	8	38957	**1.28E-01**	2.56E+07	3.29E-06	2074	**2.45E-02**	5.42E+03	1.18E-05
F9	16	37873	1.64E-01	1.24E+07	4.34E-06	2775	4.95E-02	2.78E+03	1.79E-05
F9	32	39944	2.58E-01	9.79E+06	6.46E-06	5423	1.08E-01	2.07E+03	1.99E-05
F11	8	9231	**8.35E-02**	4.30E+06	9.05E-06	27127	5.84E-02	4.00E+02	2.15E-06
F11	16	4688	9.50E-02	1.85E+06	2.03E-05	1003	**3.32E-02**	1.00E+03	3.31E-05
F11	32	4440	1.95E-01	2.45E+06	4.38E-05	400	3.74E-02	2.71E+04	9.33E-05

identifier as shown in Table 1 and Table 2, and s is the population size). On the other hand, the speedup obtained by TLBO with regard to Jaya, in terms of total execution time for each function evaluated with different population size, is shown in Fig. 5; a value in this index greater than 1 in the Y axis indicates that TLBO outperforms Jaya; in the X axis, the same identifiers (*Fi-s*) are used for the function evaluated and the population size as in Fig. 4. It can be observed that, in general, TLBO performs better than Jaya since the former needs a smaller number of iterations. This is a rather unexpected fact, because Jaya was a priori expected to have a better performance than TLBO due to the fact that the latter has two stages (Teacher and Learner phases) instead of one and, moreover, it includes some extra operations each iteration, such as the calculation of the mean value for each design variable. This fact is reflected in the iteration time, since Jaya iterations are most times faster than TLBO iterations. However, the higher complexity of TLBO seems to help achieving a faster convergence rate. On the other hand, it is worthwhile mentioning that Jaya performance is similar to the one of TLBO when dealing with a small number of variables (function F4 Matyas and F5 Easom). Indeed, Jaya performs better in case of function F4 for the different population sizes tested. In general, the higher TLBO speedup is obtained when optimizing functions using large populations. Another relevant result comes from analyzing the relation between time and population: although it could be expected to achieve faster convergence with a larger number of individuals, the results contradict this assumption, since both algorithms generally achieve the required precision in a faster way with populations of 8 or 16 individuals.

In the development environment provided by NVidia for CUDA, a tool is included to help programmers extract information about program running, running traces, and the status of GPU resources. This tool is NVidia Visual Profiler. With NVidia Visual Profiler, GPU programmers can access valuable information about the status of the GPU and its resources while running parallel programs, which helps guide

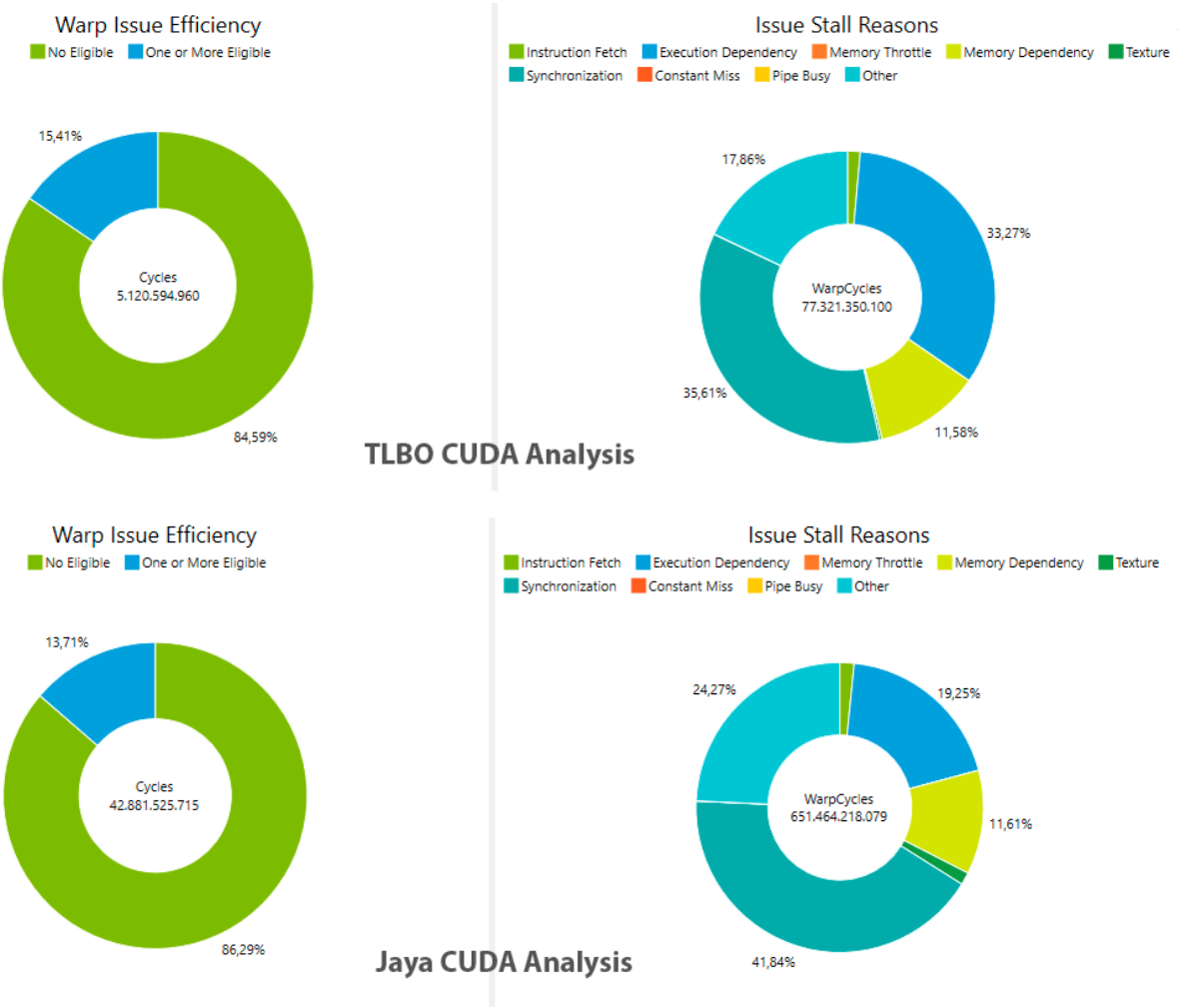

FIGURE 6. Analysis of CUDA performance obtained by TLBO (upper) and Jaya (lower).

them through potential problems or bottlenecks (especially in memory usage, blockages, and GPU occupancy). The CUDA analysis of the performance of parallel Jaya and TLBO using NVidia Visual Profiler is shown in Fig. 6, with regard to the efficiency of the warps (a warps is a set of 32 threads which run synchronously) and the distribution of stalls within the execution of each thread. With regard to the efficiency, the values of both algorithms are similar, exceeding 84% in both cases. With regard to the stall distribution, it can be observed that TLBO has a much greater dependency on the execution itself than Jaya (33.27% versus 19.25%), while Jaya spends a higher percentage of time on the synchronization of threads than that of TLBO (41.84% versus 35.61%). Moreover, it can be observed that the memory dependencies percentage is similar for both algorithms (about 11.5%).

V. CONCLUSION

In this work, a comparison of Jaya and TLBO, two recent optimization algorithms, is carried out. Both algorithms are implemented in a manycore implementation using CUDA on a GPU platform. For comparing the implementations five unconstrained functions, which are part of a well-known benchmark, are used. The only parameters which need to be set for both algorithms are population size, number of iterations, and number of runs. The indices calculated so as to compare the implementations of Jaya and TLBO were the iterations needed to find a valid solution and the time spent for performing such number of iterations.

As expected, results show that the iteration time of TLBO if higher than that of Jaya. This fact is due to the higher complexity of TLBO, since each iteration of the algorithm implies two stages whereas Jaya only performs one stage; moreover, TLBO has more time-consuming operations such as computing the mean of each variable design. However, with regard to the average time needed to find a valid solution, TLBO is generally faster than Jaya. That is, TLBO requires fewer iterations than Jaya to converge in an admissible solution. Therefore, although Jaya iterations are faster than the ones of TLBO, the higher complexity of the latter seems to be a major factor to help achieving an optimal solution by using

much fewer iteration, so the overall time required is usually higher in case of Jaya. On the other hand, it is worthwhile mentioning that the specific features of the function to be optimized (mainly the number of design variables) have a strong influence in the speed of the algorithm with regard to achieving a required precision: functions with a high number of variables are more suitable to be optimized by TLBO, while functions with a small number of variables are more suitable to be optimized with Jaya. Finally, a conclusion that can be applied to both algorithms comes from observing the results related to the different population sizes tested: in general, larger populations (32 individuals) do not lead to a faster convergence, but most of the time a population of 16 or even 8 individuals is enough to achieve the required precision.

As a future work, new benchmark functions used in other works should be added to follow with a deeper comparison of Jaya and TLBO. Furthermore, there are new versions of the Jaya and TLBO algorithms improving their results. Therefore, it will be very interesting to create a CUDA parallel version of these improvements of both algorithms and to compare them with the older versions. Finally, another interesting work could be to perform parallel implementations of Jaya and TLBO addressed to other parallelism techniques or platforms and to compare them with the CUDA implementations of Jaya and TLBO proposed in this work.

REFERENCES

[1] A. J. Umbarkar, M. S. Joshi, and P. D. Sheth, "OpenMP dual population genetic algorithm for solving constrained optimization problems," *Int. J. Inf. Eng. Electron. Bus.*, vol. 7, no. 1, pp. 59–65, 2015.

[2] R. Baños, J. Ortega, and C. Gil, "Comparing multicore implementations of evolutionary meta-heuristics for transportation problems," *Ann. Multicore GPU Program.*, vol. 1, no. 1, pp. 9–17, 2014.

[3] R. Baños, J. Ortega, and C. Gil, "Hybrid MPI/OpenMP parallel evolutionary algorithms for vehicle routing problems," in *Proc. Eur. Conf. Appl. Evol. Comput.*, Granada, Spain, 2014, pp. 653–664.

[4] P. Delisle, M. Krajecki, M. Gravel, and C. Gagné, "Parallel implementation of an ant colony optimization metaheuristic with OPENMP," in *Proc. 3rd Eur. Workshop OpenMP (EWOMP)*, Barcelona, Spain, 2001, pp. 8–12.

[5] Y. Tan and K. Ding, "A survey on GPU-based implementation of swarm intelligence algorithms," *IEEE Trans. Cybern.*, vol. 46, no. 9, pp. 2028–2041, Sep. 2016.

[6] G.-H. Luo, S.-K. Huang, Y.-S. Chang, and S.-M. Yuan, "A parallel Bees Algorithm implementation on GPU," *J. Syst. Archit.*, vol. 60, pp. 271–279, Mar. 2014.

[7] A. Delévacq, P. Delisle, M. Gravel, and M. Krajecki, "Parallel ant colony optimization on graphics processing units," *J. Parallel Distrib. Comput.*, vol. 73, pp. 52–61, Jan. 2013.

[8] L. Mussi, F. Daolio, and S. Cagnoni, "Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture," *Inf. Sci.*, vol. 181, pp. 4642–4657, Oct. 2011.

[9] L. de P. Veronese and R. A. Krohling, "Differential evolution algorithm on the GPU with C-CUDA," in *Proc. IEEE Congr. Evol. Comput.*, Jul. 2010, pp. 1–7.

[10] Y. Zhou and Y. Tan, "GPU-based parallel particle swarm optimization," in *Proc. IEEE Congr. Evol. Comput.*, May 2009, pp. 1493–1500.

[11] R. V. Rao and V. Patel, "An elitist teaching-learning-based optimization algorithm for solving complex constrained optimization problems," *Int. J. Ind. Eng. Comput.*, vol. 3, no. 4, pp. 535–560, Jul. 2012.

[12] R. V. Rao and V. Patel, "Comparative performance of an elitist teaching-learning-based optimization algorithm for solving unconstrained optimization problems," *Int. J. Ind. Eng. Comput.*, vol. 4, no. 1, pp. 29–50, 2013.

[13] R. V. Rao, V. J. Savsani, and D. P. Vakharia, "Teaching-learning-based optimization: A novel method for constrained mechanical design optimization problems," *Comput.-Aided Des.*, vol. 43, no. 3, pp. 303–315, Mar. 2011.

[14] R. V. Rao and V. Patel, "An improved teaching-learning-based optimization algorithm for solving unconstrained optimization problems," *Scientia Iranica*, vol. 20, no. 3, pp. 710–720, 2013.

[15] A. Jimeno-Morenilla, J. L. Sánchez-Romero, H. Migallón, and H. Mora-Mora, "Jaya optimization algorithm with GPU acceleration," *J. Supercomput.*, vol. 75, pp. 1094–1106, Mar. 2019.

[16] R. V. Rao, "Jaya: A simple and new optimization algorithm for solving constrained and unconstrained optimization problems," *Int. J. Ind. Eng. Comput.*, vol. 7, no. 1, pp. 19–34, 2016.

[17] M. Ebraheem and T. R. Jyothsna, "Comparative performance evaluation of teaching learning based optimization against genetic algorithm on benchmark functions," in *Proc. IEEE Power, Commun. Inf. Technol. Conf. (PCITC)*, Oct. 2015, pp. 327–331.

[18] S. R. Shah and S. B. Takmare, "A review of methodologies of TLBO algorithm to test the performance of benchmark functions," *Program. Device Circuits Syst.*, vol. 9, no. 7, pp. 141–145, 2017.

[19] 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 *Proc. IEEE Power and Energy Soc. Gen. Meeting (PESGM)*, Jul. 2016, pp. 1–5.

[20] K. Abhishek, V. R. Kumar, S. Datta, and S. S. Mahapatra, "Application of JAYA algorithm for the optimization of machining performance characteristics during the turning of CFRP (epoxy) composites: Comparison with TLBO, GA, and ICA," *Eng. Comput.*, vol. 33, pp. 457–475, Jul. 2017.

[21] M. Bhoye, M. H. 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 *Proc. Int. Conf. Energy Efficient Technol. Sustainability (ICEETS)*, Apr. 2016, pp. 497–502.

[22] 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 *Proc. 2nd Int. Conf. Adv. Elect. Electron. Inf. Commun. Bio-Inform. (AEEICB)*, Feb. 2016, pp. 224–228.

[23] S. Mishra and P. K. Ray, "Power quality improvement using photovoltaic fed DSTATCOM based on JAYA optimization," *IEEE Trans. Sustain. Energy*, vol. 7, no. 4, pp. 1672–1680, Oct. 2016.

[24] A. J. Umbarkar, N. M. Rothe, and A. S. Sathe, "OpenMP teaching-learning based optimization algorithm over multi-core system," *Int. J. Intell. Syst. Appl.*, vol. 7, no. 7, pp. 57–65, 2015.

[25] R. V. Rao and G. G. Waghmare, "A new optimization algorithm for solving complex constrained design optimization problems," *Eng. Optim.*, vol. 49, no. 1, pp. 60–83, 2016.

[26] X. He, J. Huang, Y. Rao, and L. Gao, "Chaotic teaching-learning-based optimization with Lévy flight for global numerical optimization," *Comput. Intell. Neurosci.*, vol. 2016, p. 43, Jan. 2016, Art. no. 8341275.

[27] J. Yu, C. H. Kim, A. Wadood, T. Khurshiad, and S.-B. Rhee, "A novel multi-population based chaotic JAYA algorithm with application in solving economic load dispatch problems," *Energies*, vol. 11, no. 8, p. 1946, 2018.

[28] K. V. Rao, "Power consumption optimization strategy in micro ball-end milling of D2 steel via TLBO coupled with 3D FEM simulation," *Measurement*, vol. 132, pp. 68–78, Jan. 2019.

[29] Z. Nadeem, N. Javaid, A. W. Malik, and S. Iqbal, "Scheduling appliances with GA, TLBO, FA, OSR and their hybrids using chance constrained optimization for smart homes," *Energies*, vol. 11, no. 4, p. 888. 2018.

[30] A. Mishra and D. Shrivastava, "A TLBO and a Jaya heuristics for permutation flow shop scheduling to minimize the sum of inventory holding and batch delay costs," *Comput. Ind. Eng.*, vol. 124, pp. 509–522, Oct. 2018.

[31] R. V. Rao, H. S. Keesari, P. Oclon, and J. Taler, "An adaptive multi-team perturbation-guiding Jaya algorithm for optimization and its applications," *Eng. Comput.*, pp. 1–29, 2019. doi: 10.1007/s00366-019-00706-3.

[32] S. Xu, Y. Wang, and Z. Wang, "Parameter estimation of proton exchange membrane fuel cells using eagle strategy based on JAYA algorithm and Nelder-Mead simplex method," *Energy*, vol. 173, pp. 457–467, Apr. 2019.

[33] R. V. Rao and H. S. Keesari, "Multi-team perturbation guiding Jaya algorithm for optimization of wind farm layout," *Appl. Soft Comput.*, vol. 71, pp. 800–815, Oct. 2018.

[34] R. V. Rao and A. Saroj, "A self-adaptive multi-population based Jaya algorithm for engineering optimization," *Swarm Evol. Comput.*, vol. 37, pp. 1–26, Dec. 2017.

[35] M. Gambhi and and S. Gupta, "Advanced optimization algorithms for grating based sensors: A comparative analysis," *Optik*, vol. 164, pp. 567–574, Jul. 2018.
[36] S. Ghavidel, A. Azizivahed, and L. Li, "A hybrid Jaya algorithm for reliability-redundancy allocation problems," *Eng. Optim.*, vol. 50, no. 4, pp. 698–715, 2018.
[37] R. V. Rao, A. Saroj, and S. Bhattacharyya, "Design optimization of heat pipes using elitism-based self-adaptive multipopulation Jaya algorithm," *J. Thermophys. Heat Transf.*, vol. 32, no. 3, pp. 702–712, 2018.
[38] L. Wang, C. Huang, and L. Huang, "Parameter estimation of the soil water retention curve model with Jaya algorithm," *Comput. Electron. Agricult.*, vol. 151, pp. 349–353, Aug. 2018.
[39] S. S. Ramanlal and S. S. Krishna, "GPU implementation of TLBO algorithm to test constrained and unconstrained benchmark functions," in *Proc. Int. Conf. Comput. Anal. Secur. Trends (CAST)*, Dec. 2016, pp. 544–549,
[40] S. Xiao and W.-C. Feng, "Inter-block GPU communication via fast barrier synchronization," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, Apr. 2010, pp. 1–12.
[41] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-M. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proc. 13th SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2008, pp. 73–82.

H. RICO-GARCIA was born in Alicante, Spain, in 1982. He received the degree in computer systems engineering and the master's degree in information technology from the University of Alicante, Spain, where he is currently pursuing the Ph.D. degree. His research interests include high-performance computer architecture, embedded systems, the Internet of Things, and cloud computing paradigm.

JOSE-LUIS SANCHEZ-ROMERO was born in Elche, Spain, in 1970. He received the Ph.D. degree from the University of Alicante, Spain, in 2009, where he is currently a tenured Associate Professor with Computer Technology Department. He also has relevant experience in the research of computer science teaching improvement. His research interests include high-performance computer architecture, computer arithmetic, and CAD/CAM systems.

A. JIMENO-MORENILLA was born in Spain, in 1970. He received the Ph.D. degree from the University of Alicante, Spain, in 2003, where he is currently a Full Professor with Computer Technology Department. He also has considerable experience in the investigation of professional skills of computer engineers. His research interests include computational geometry for design and manufacturing, rapid and virtual prototyping, and high-performance computer architectures.

H. MIGALLON-GOMIS received the degree in physics and the Electronic Engineering degree from the University of Valencia, and the Ph.D. degree, in 2005. He is currently an Associate Professor with the Computer Engineering Department, Miguel Hernández University, Elche, Spain. He is a member of the "Architecture and Computer Technology" Research Group, Miguel Hernández University, and the "High Performance Computing and Parallelism" Research Group, University of Alicante. His main research interests include parallel algorithms for solving linear and nonlinear systems, parallel algorithms for image and video processing, parallel heuristic optimization algorithms, and parallel high-level interfaces for heterogeneous platforms.

H. MORA-MORA received the Ph.D. degree in computer science from the University of Alicante, Spain, in 2003, where he is currently an Associate Professor with Computer Technology Department. His research interests include computer modeling, computer architectures, high-performance computing, embedded systems, the Internet of Things, and cloud computing paradigm.

R. V. RAO is currently a Professor of mechanical engineering and the Dean (Faculty Welfare) of the Sardar Vallabhbhai National Institute of Technology, Surat, India. He has about 28 years of teaching and research experience. He has authored more than 350 research articles published in various international journals and conference proceedings. His research interests include advanced engineering optimization techniques and their applications to the problems of design, thermal, and manufacturing engineering. He is also on the editorial boards of various international journals.