A Massively Parallel BWP Algorithm for Solving Large-Scale Systems of Nonlinear Equations

Bruno Silva
PhD Program in Computer Eng., Univ. of Madeira
SRE, Regional Government of Madeira
Funchal, Madeira Is., Portugal
bruno.silva@madeira.gov.pt

Luiz Guerreiro Lopes
Faculty of Exact Sciences and Engineering
University of Madeira
Funchal, Madeira Is., Portugal
lopes@uma.pt

***Abstract*—This paper presents a GPU-based massively parallel implementation of the Best-Worst-Play (BWP) metaphor-less optimization algorithm, which results from the combination of two other simple and quite efficient population-based algorithms, Jaya and Rao-1, that have been used to solve a variety of problems. The proposed parallel GPU version of the algorithm is here used for solving large nonlinear equation systems, which have enormous importance in different areas of science, engineering, and economics and are usually considered the most difficult class of problems to solve by traditional numerical methods. The proposed parallelization of the BWP algorithm was implemented using the Julia programming language on a GeForce RTX 3090 GPU with 10 496 CUDA cores and 24 GB of VRAM and tested on a set of challenging scalable systems of nonlinear equations with dimensions between 500 and 2000. Depending on the tested problem and dimension, the GPU-based implementation of BWP exhibited a speedup up to 283.17×, with an average of 161.21×, which shows the efficiency of the proposed GPU-based parallel version of the BWP algorithm.**

***Index Terms*—metaheuristic optimization, metaphor-less algorithms, GPU acceleration, best-worst-play algorithm, nonlinear equation systems**

I. Introduction

Metaheuristic optimization provides a framework for solving complex optimization problems in which population-based optimization algorithms are included. These algorithms rely on a population of candidate solutions to guide the search process and effectively explore different regions of the search space. These types of methods stand out for their adaptability, ability, and versatility to solve a variety of optimization problems without requiring prior knowledge about the problem specifics.

The Best-Worst-Play (BWP) algorithm [1] is a metaphor-less population-based optimization technique that can be used for approximating solutions of systems of nonlinear equations (SNLEs). These problems can be transformed into nonlinear minimization problems by using the sum of the absolute values of the residuals as the objective function.

As solving SNLEs is difficult and requires substantial computational resources, which can increase rapidly with problem size, these are excellent candidates to be processed by massively parallel algorithms.

This paper proposes a parallel implementation of BWP that takes advantage of graphics processing unit (GPU) hardware to perform general-purpose parallel computation.

The GPU-based implementation of the algorithm is also compared with the sequential version, running on a central processing unit (CPU), in the resolution of a set of scalable and difficult-to-solve SENLs selected as test problems in order to assess and analyze different performance attributes.

II. Best-Worst-Play Algorithm

Best-Worst-Play (BWP) [1] is a metaphor-less algorithm motivated by the simplicity and efficiency of the Jaya [2] and Rao-1 [3] algorithms. Through the hybridization of both of these algorithms, in which they are executed sequentially, one after the other, BWP provides a reasonable balance of exploration and exploitation. This means that BWP is able to maintain the same straightforward approach as Jaya and Rao-1, i.e., by focusing on identifying the best and worst candidate solutions within a given population, it is able to gradually increase the quality of the population and converge to a near-optimal solution.

As parameter-less algorithms, BWP, Jaya, and Rao-1 only require common optimization control parameters such as the maximum number of possible iterations ($maxIter$), the population size ($popSize$), and the number of decision variables ($numVar$) to maximize or minimize a given objective function $f(X)$.

Taking into consideration a design variable index v ranging from 1 to $numVar$, a population index p varying from 1 to $popSize$, and the iteration index i ranging from 1 to $maxIter$, let $X_{v,p,i}$ be the value of the v-th variable of the p-th population candidate during the i-th iteration. The updated value $X_{v,p,i}^{new}$ for the Jaya algorithm is determined as:

$$\begin{aligned} X_{v,p,i}^{new} = X_{v,p,i} + r_{1,v,i}\left(X_{v,best,i} - \left|X_{v,p,i}\right|\right) - \\ r_{2,v,i}\left(X_{v,worst,i} - \left|X_{v,p,i}\right|\right), \end{aligned} \tag{1}$$

where $r_{1,v,i}$ and $r_{2,v,i}$ are uniformly distributed random numbers between 0 and 1, and $X_{v,best,i}$ and $X_{v,worst,i}$ are the population candidates with the best and worst fitness values.

The Rao-1 algorithm employs a similar approach to the Jaya principles, in which the population avoids the worst solution while moving towards the best, but in a more simplified manner, as shown in the following equation:

$$X_{v,p,i}^{new} = X_{v,p,i} + r_{1,v,i}\left(X_{v,best,i} - X_{v,worst,i}\right) \tag{2}$$

```
/* Initialization */
Initialize $numVars$, $popSize$ and $maxIters$;
Generate initial population $X$;
Evaluate fitness value $f(X)$;
$i \leftarrow 1$;
/* Main loop */
while $i \leq maxIters$ do
    Determine $X_{v,best,i}$ and $X_{v,worst,i}$;
    for $p \leftarrow 1, popSize$ do
        for $v \leftarrow 1, numVars$ do
            Update population $X^{new}_{v,p,i}$ by Eq. (3);
        end for
        Calculate $f(X^{new}_{v,p,i})$;
        if $f(X^{new}_{v,p,i})$ is better than $f(X_{v,p,i})$ then
            $X_{v,p,i} \leftarrow X^{new}_{v,p,i}$;
            $f(X_{v,p,i}) \leftarrow f(X^{new}_{v,p,i})$;
        else
            Keep $X_{v,p,i}$ and $f(X_{v,p,i})$ values;
        end if
    end for
    Determine $X_{v,best,i}$ and $X_{v,worst,i}$;
    for $p \leftarrow 1, popSize$ do
        for $v \leftarrow 1, numVars$ do
            Update population $X^{new}_{v,p,i}$ by Eq. (4);
        end for
        Calculate $f(X^{new}_{v,p,i})$;
        if $f(X^{new}_{v,p,i})$ is better than $f(X_{v,p,i})$ then
            $X_{v,p,i} \leftarrow X^{new}_{v,p,i}$;
            $f(X_{v,p,i}) \leftarrow f(X^{new}_{v,p,i})$;
        else
            Keep $X_{v,p,i}$ and $f(X_{v,p,i})$ values;
        end if
    end for
    $i \leftarrow i + 1$;
end while
Report best solution found;
```

Fig. 1. Sequential BWP algorithm.

The search strategy for the BWP algorithm is based on both the Jaya and Rao-1 formulations, and as such, the new modified value ($X^{new}_{v,p,i}$) is determined by two equations, as follows:

$$X^{new}_{v,p,i} = X_{v,p,i} + r_{1,v,i}\left(X_{v,best,i} - |X_{v,p,i}|\right) - r_{2,v,i}\left(X_{v,worst,i} - |X_{v,p,i}|\right) \tag{3}$$

$$X^{new}_{v,p,i} = X_{v,p,i} + r_{3,v,i}\left(X_{v,best,i} - |X_{v,worst,i}|\right) \tag{4}$$

Both equations above are applied in sequence to all candidate solutions along all the iterations, as shown in the algorithm of Fig. 1.

III. GPU Parallelization of the BWP Algorithm

To take full advantage of the power of GPUs for general-purpose computing and achieve significantly improved performance and efficiency over traditional CPU-based implementations, it is necessary to resort to a parallel computing platform and programming model. In this implementation, the Compute Unified Device Architecture (CUDA) platform was used with the aim of massively parallelizing the BWP algorithm. The CUDA software stack was developed by NVIDIA as a way to abstract the complexities of GPU programming and enable communication and coordination between the CPU and GPU.

The first step in executing a massively parallel implementation of any algorithm on the GPU is to carry out a deep analysis of its organization, dependencies, and data structure in order to obtain a thorough understanding and be able to determine which parts of the algorithm are able to be parallelized. In CUDA, these parallel computational units are called kernel functions.

The CUDA thread hierarchy organizes multiple threads into a thread block, which in turn is part of a grid of other thread blocks that are used by a kernel to process data. This grid can be composed of one-dimensional (1D), two-dimensional (2D), or three-dimensional (3D) thread blocks, depending on the algorithm requirements and data structure. Figure 2 illustrates the indexing of a 2D data structure into a 2D thread hierarchy (i.e., a grid).

CUDA uses a thread indexing mechanism to control which specific data pieces should be accessible by each individual. In general terms, this mechanism handles the mapping of data to individual threads. In Fig. 2, it is also possible to see the mapping of a population of possible candidates to a CUDA kernel. Data is divided into a tile of blocks, composed by a number of threads, within a grid, and each element is indexed, meaning that there is a mechanism to uniquely identify each individual component using a set of built-in variables.

The upper right corner of Fig. 2 illustrates how the data-to-thread indexing of the population information is performed in the GPU-based BWP algorithm. Data from candidate 9 and variable 6 (considering that indexing starts at 0) is mapped to be processed by the thread (4,3) from the block (1,1). This is how computation over the population data can be performed in parallel on multiple blocks of threads. For the parallel processing of data associated with the cost function, a 1D thread block arrangement is utilized, which means that the process is similar but employs one-dimensional thread blocks to perform the data-to-thread indexing.

Determining the optimal number of blocks and threads per block is a key performance factor as it establishes the GPU hardware occupancy, i.e., the allocation of the GPU computing resources. The maximum number of threads that can run concurrently and the number of threads allowed per block depend on the GPU hardware used, which is limited by characteristics such as the quantity of multiprocessing units available and the number of threads per multiprocessor. In order to achieve the most effective ratio, this process typically requires complex computation to deal with GPU specifications as well as kernel properties and restrictions. The implementation of the GPU-based BWP algorithm addresses this issue by utilizing a set of new routines introduced in CUDA version 6.5 to address the

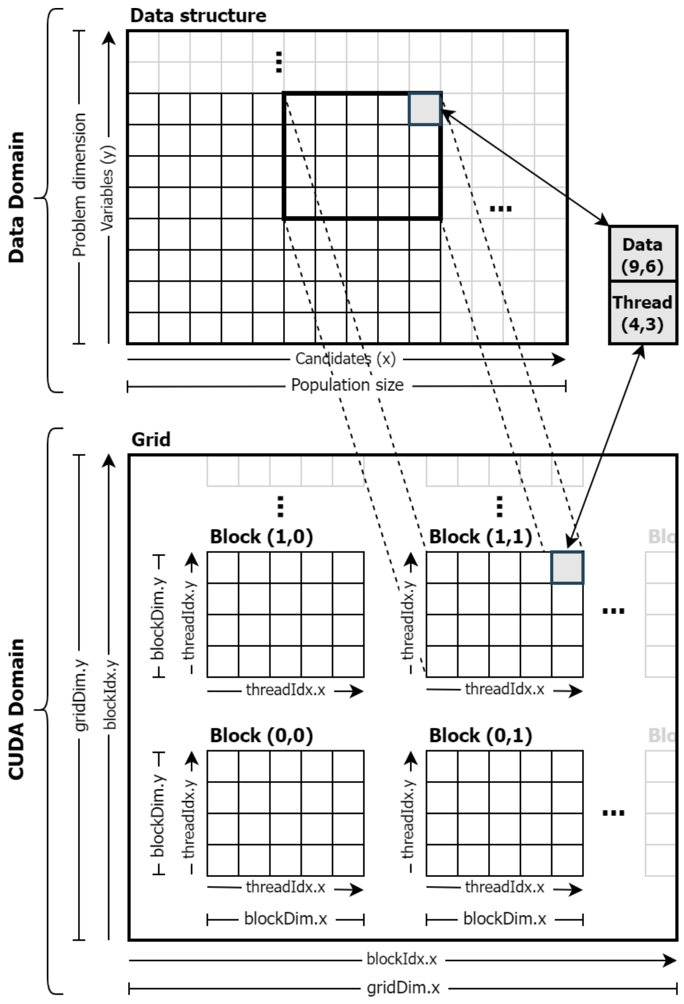

Fig. 2. 2D CUDA thread hierarchy for data and thread indexing.

```
 1: /* Initialization */
 2: Initialize $numVar$, $popSize$ and $maxIter$;                ▷ Host
 3: $X \leftarrow$ GENERATE_INITIAL_POP_KERNEL();                ▷ Device
 4: EVALUATE_FITNESS_KERNEL($X$);
 5: $i \leftarrow 1$;                                            ▷ Host
 6: /* Main loop */
 7: while $i \leq maxIter$ do                                    ▷ Host
 8:     /* Apply Eq. (3) */
 9:     Determine $X_{best,i}$ and $X_{worst,i}$;                ▷ Device
10:     $X_i^{new} \leftarrow$ UPDATE_POP_KERNEL($X_i$, ‘Eq_a’);
11:     EVALUATE_FITNESS_KERNEL($X_i^{new}$);
12:     $X_i \leftarrow$ SELECT_BEST_KERNEL($X_i, X_i^{new}$);
13:     /* Apply Eq. (4) */
14:     Determine $X_{best,i}$ and $X_{worst,i}$;                ▷ Device
15:     $X_i^{new} \leftarrow$ UPDATE_POP_KERNEL($X_i$, ‘Eq_b’);
16:     EVALUATE_FITNESS_KERNEL($X_i^{new}$);
17:     $X_i \leftarrow$ SELECT_BEST_KERNEL($X_i, X_i^{new}$);
18:     $i \leftarrow i + 1$;                                    ▷ Host
19: end while
20: Report best solution found;                                  ▷ Device and Host
```

Fig. 3. GPU-based parallel BWP algorithm.

occupancy calculation and launch configurations of kernels. This resulted in a kernel launch configuration function that is able to dynamically propose the optimal number of threads and blocks required to achieve maximum occupancy for the specific kernel and GPU hardware used. In this way, it was possible to design kernels where the main focus is on data and computation characteristics, which are able to run on and adapt to different GPU hardware and are inherently designed to scale horizontally by utilizing additional threads, and deliver improved performance on more modern hardware without requiring codebase modifications.

The proposed massively parallel GPU-based BWP implementation is presented in the algorithm of Fig. 3. This implementation uses a heterogeneous computing approach, meaning that the algorithm uses both the CPU (the host) and the GPU (the device) during the entire computational process.

The CPU manages the overall execution of the algorithm by carrying out sequential workloads like initializing the GPU hardware, memory allocations, data transfers, and handling synchronizations between CPU and GPU (tasks identified in the algorithm of Fig. 3 with the comment *Host*). On the other hand, the GPU handles all parts of the algorithm that could be parallelized. The GPU parallel tasks are identified in Fig. 3 with the comment *Device* or all functions whose names end in *kernel*.

Having parallel execution divided among multiple kernels simplifies the management of thread synchronization because all processed data is synchronized within the device's global memory when one kernel completes its execution and the subsequent kernel starts.

Being a population-based optimization algorithm, BWP has to maintain and update information about a population (a matrix of $popSize \times numVars$ size), the fitness value of each population candidate (an array of $popSize$ length) and information about the best and worst candidate solutions (arrays of length $numVars$). Due to the performance costs associated with transferring data between the host and the device, the number of data transfers has to be as minimal as possible, meaning that moving or synchronizing data structures repeatedly would result in an adverse effect on the computational performance.

Besides transferring control parameters like the $numVars$ and $popSize$, all remaining data for the BWP algorithm is generated or computed directly on the device. Data is stored direct in the device global memory and the only data transfer between device and host is the best solution found at the end of the execution. This data locality strategy resulted in a very efficient reduction of data transfers, with no data transfers during the main loop, which is the most computationally costly phase of the entire implementation, with the exception of two variables in the initialization phase and the best solution found at the end of the algorithm.

```
1:  function UPDATE_POP_KERNEL($X_i, Equation$)
2:      /* Device code */
3:      Determine $row$ using $blockDim.x$, $blockIdx.x$, and $threadIdx.x$;
4:      Determine $col$ using $blockDim.y$, $blockIdx.y$, and $threadIdx.y$;
5:      if $row \leq popSize$ and $col \leq numVar$ then
6:          if Equation = ‘Eq_a’ then                              ▷ Eq. (3)
7:              $X_i^{new}[row, col] = X_i[row, col] + rand() \times (X_{best,i}[col] - |X_i[row, col]|) - rand() \times (X_{worst,i}[col] - |X_i[row, col]|)$;
8:          else if Equation = ‘Eq_b’ then                         ▷ Eq. (4)
9:              $X_i^{new}[row, col] = X_i[row, col] + rand() \times (X_{best,i}[col] - |X_{worst,i}[col]|)$;
10:         end if
11:     end if
12: end function
```

Fig. 4. Kernel to update population.

The algorithm in Fig. 4 describes the kernel implementation for updating the population. This kernel is able to determine all the new candidate solutions for the entire population in parallel, as it leverages the CUDA thread hierarchy to handle the data and thread indexing in a 2D grid arrangement of blocks and threads, in a similar configuration to that shown in Fig. 2.

This kernel employs the variables row and col to handle data indexing. The first one refers to the population size and uses the x dimension of the thread and block data for indexing, while the second one relates to the number of variables (i.e., the problem dimension) and indexes using the y dimension of the thread and block data.

IV. Computational Experiments

A total of 10 scalable and hard-to-solve SNLEs were selected from the literature to be used as test problems. These problems are listed in Table I along with their domain D and additional parameters used.

With the goal of finding out how well the sequential and parallel versions of the BWP algorithm work for large-scale SNLEs, problem sizes of 500, 1000, 1500, and 2000 were used for testing. The population size was set at 10× the problem dimension, resulting in 5000, 10 000, 15 000, and 20 000, respectively.

The number of iterations was fixed at 1000, as the objective of the experiments is to compare the efficiency of the parallel version of BWP to that of the corresponding sequential algorithm, and not to focus on how well the algorithm performs with this class of problems considered. For each combination of algorithm, test problem, and dimension, a total of 51 independent runs were performed.

The sequential experiments were executed using an Intel Core i7-5700HQ CPU running at 2.70 GHz up to 3.50 GHz, with 4 cores and 8 threads and 16 GB RAM. The GPU parallel experiments were performed using a GeForce RTX 3090 GPU with 10 496 CUDA cores and 24 GB GDDR6X VRAM.

TABLE I
Test problems

No.	Problem name, domain, and parameters	Ref.
1	Broyden tridiagonal function $D = ([-1, 1], \ldots, [-1, 1])^T$	[4]
2	Discrete boundary value function $D = ([0, 5], \ldots, [0, 5])^T$ $h = \frac{1}{n+1}$, $t_i = ih$	[4]
3	Modified Rosenbrock function $D = ([-10, 10], \ldots, [-10, 10])^T$	[5]
4	Powell badly scaled function $D = ([0, 100], \ldots, [0, 100])^T$	[5]
5	The beam problem $D = ([-100, 100], \ldots, [-100, 100])^T$ $\alpha = 11$, $h = \frac{1}{n+1}$	[6]
6	The Bratu problem $D = ([-100, 100], \ldots, [-100, 100])^T$ $\alpha = 3.5$, $h = \frac{1}{n+1}$	[6]
7	Extended Rosenbrock function $D = ([-100, 100], \ldots, [-100, 100])^T$	[4]
8	Schubert–Broyden function $D = ([-100, 100], \ldots, [-100, 100])^T$	[7]
9	Extended Powell singular function $D = ([-5, 5], \ldots, [-5, 5])^T$	[4]
10	Martínez function $D = ([-100, 100], \ldots, [-100, 100])^T$	[8]

Note: For all the problems, $n = 500, 1000, 1500, 2000$.

Both the sequential and parallel versions of the BWP algorithm were implemented using the Julia programming language (version 1.9.0) and double-precision floating-point arithmetic.

V. Results and Discussion

The execution time of each individual run of the sequential and parallel implementations of the BWP algorithm was measured, and the average result for each experimental evaluation is presented in Table II.

The mean CPU and GPU times presented in Table II correspond to the average result of 51 independent runs of 1000 iterations each and cover the entire algorithm, i.e., the initialization of parameters, the generation of the initial population, the main loop, and the reporting of the best solution found at the end of each run. In the case of the parallel algorithm, this also implies that all costs associated with data movement are taken into account.

Data shows that the GPU-based algorithm was more efficient when compared to its sequential implementation, having been consistently the fastest in terms of execution time. Considering the sequential version of the algorithm as a baseline, the execution time improvement (speedup) obtained by the GPU-based implementation ranged from 69.89× to 283.17× and averaged 161.21×, depending on the problem and dimension tested.

When averaging the results by problem dimension (see Table III), the results show that the GPU-based algorithm gets

TABLE II
COMPUTATIONAL RESULTS

Prob. dim.	Pop. size	Test prob.	Mean CPU time (s)	Mean GPU time (s)	Mean speedup
500	5000	1	136.1736	1.7823	76.40
		2	149.0208	2.1322	69.89
		3	192.5926	2.0338	94.70
		4	184.3982	2.0084	91.81
		5	157.8388	1.9124	82.53
		6	158.2328	1.9063	83.00
		7	199.0742	1.2277	162.16
		8	190.7176	1.1884	160.48
		9	165.0380	1.2728	129.67
		10	149.3742	1.1875	125.79
1000	10 000	1	414.3892	2.8573	145.03
		2	434.6956	4.1474	104.81
		3	553.4086	4.1180	134.39
		4	685.6960	4.2553	161.14
		5	504.6524	4.4473	113.47
		6	493.7792	4.3857	112.59
		7	589.2004	2.4716	238.39
		8	479.4048	2.4645	194.52
		9	474.0440	2.7117	174.82
		10	378.2830	2.5341	149.28
1500	15 000	1	814.2546	4.5280	179.83
		2	861.8682	6.6186	130.22
		3	1158.4782	6.5886	175.83
		4	1413.0258	7.4532	189.59
		5	1055.8556	8.0191	131.67
		6	1196.2840	7.9709	150.08
		7	1151.6798	4.0671	283.17
		8	985.9276	4.1533	237.38
		9	946.2958	4.3409	217.99
		10	766.8736	4.3180	177.60
2000	20 000	1	1324.9416	6.7269	196.96
		2	1337.9348	10.1912	131.28
		3	2049.7252	10.0461	204.03
		4	2490.5636	11.8766	209.70
		5	1756.7964	13.0952	134.16
		6	1756.3878	12.8975	136.18
		7	1650.8750	6.0108	274.65
		8	1454.2976	6.1807	235.30
		9	1489.4332	6.3099	236.05
		10	1368.4750	6.4652	211.67

faster (i.e., the speedup increases) as the problem dimension grows larger. This indicates that the implemented parallelization strategy is very efficient and is able to successfully deal with the increase in computational complexity.

TABLE III
MEAN COMPUTATIONAL RESULTS PER PROBLEM DIMENSION

Prob. dim.	Pop. size	Mean CPU time (s)	Mean GPU time (s)	Mean speedup
500	5000	168.2461	1.6652	107.64
1000	10 000	500.7553	3.4393	152.84
1500	15 000	1035.0543	5.8058	187.34
2000	20 000	1667.9430	8.9800	197.00

The computational scaling was evaluated by analyzing the mean execution time for each test problem while adjusting the dimension for both the sequential (see Fig. 5) and the parallel (see Fig. 6) implementations.

In the sequential version of the algorithm, all test problems exhibit a growing trend of execution time for each dimension increase, although the performance penalty for rises in the problem dimension is larger than the rate of data increase. Problems 3 and 4 present a stepper increasing slope, which indicates that these problems are the most computationally challenging, especially at the highest dimension. Looking at the parallel BWP algorithm execution time in Fig. 6, it is noticeable that the problems with numbers 2, 3, 4, 5, and 6 have a computational time growth larger than the rest of the test problems. This suggests that the implementation of these test problems resulted in a less efficient parallelization when compared with problems 1, 7, 8, 9, and 10, for which the time needed to compute the higher dimensions varies at a more constant rate.

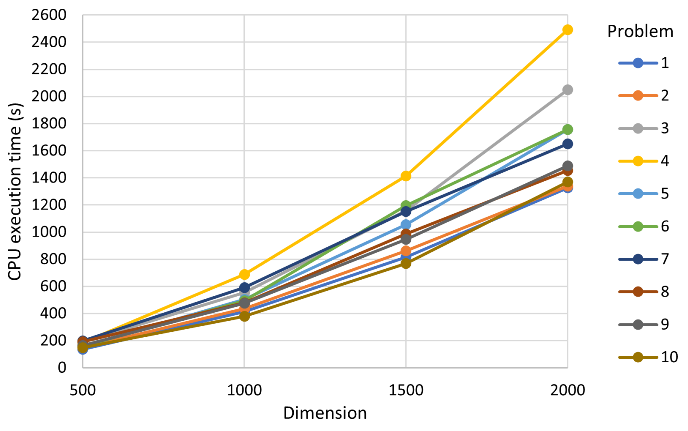

Fig. 5. CPU execution time per problem and dimension.

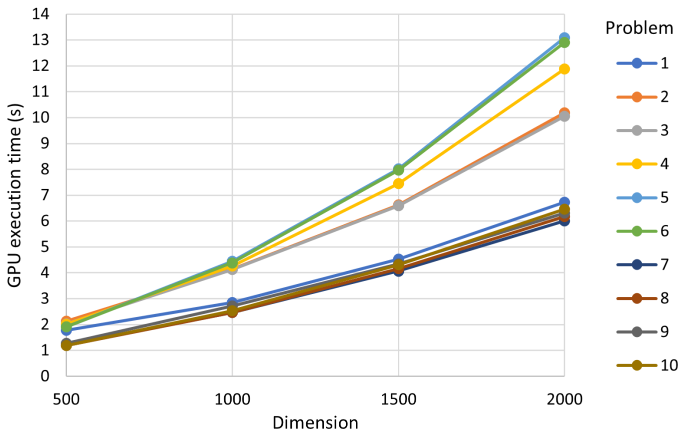

Fig. 6. GPU execution time per problem and dimension.

The scalability and efficiency of the GPU-based parallel BPW algorithm are presented in Fig. 7 in the form of the mean speedup per problem and dimension. Results show that the parallel algorithm is very effective in handling workload increases, as the mean speedup shows a positive growth as the problem sizes get larger in the majority of the test problems. This indicates effective utilization of the GPU hardware resources in order to maintain or improve performance.

The exceptions were test problems 6, 7, and 8, which in dimension 2000 failed to achieve an acceleration greater than in dimension 1500, although still accomplishing a very positive speedup gain. This could be related to underlying characteristics of the algorithm design in combination with aspects of the GPU hardware architecture used for testing (e.g., particularities with the SNLEs parallelization, combination of blocks and threads allocated, GPU memory or synchronization overheads, etc.).

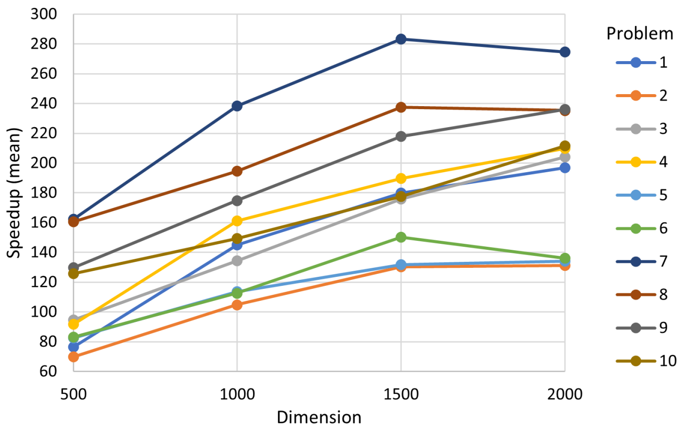

Fig. 7. Mean speedup improvement per problem and dimension.

Figure 7 also shows that the lowest speedup gains were achieved when computing with the smallest problem dimension and that generally the speedup increases along with the problem dimension. This is an expected behavior since the GPU-based algorithm is designed to utilize more threads as the problem dimension increases, meaning that the performance scalability depends both on the problem size as well as on the number of CUDA cores available in the GPU hardware.

While analyzing data between the dimensions 1500 and 2000, it is visible in Fig. 7 that problems with numbers 6, 7, and 8 show a negative slope change, i.e., a speedup regression. This means that the optimal workload point for these problems was reached around dimension 1500, as this is where the greatest speedup gain was achieved.

Within the same dimension range, the speedup for problems 2 and 5 appears to be leveling off, suggesting that the optimal workload point for these test problems is probably nearly after dimension 2000. The remaining test problems demonstrate a positive slope up to dimension 2000, implying that the parallel BWP algorithm is able to efficiently use the hardware resources to provide speedup increases as the computational complexity grows.

VI. Conclusion

The GPU-based parallel BWP algorithm was implemented using CUDA and a heterogeneous computing technique that assigned different workloads to the CPU and GPU processors. The new algorithm was tested using large and scalable SNLEs, and the results were compared with its sequential version.

Results showed that the proposed GPU-based parallel algorithm was very effective in coping with the computational complexities of solving large-scale SNLEs, achieving an average speedup of up to 283.17× over the original sequential algorithm. The parallel BWP algorithm was able to use the available GPU resources very efficiently and consistently offer performance increases with growing workloads, although in some test problems there was a small speedup regression in the largest dimension.

GPU parallelization can offer considerable performance improvements but requires a deep understanding of several paradigms, such as parallel programming, the computing platform, and the device hardware architecture, to be able to harness its processing power. The design and implementation strategy used in the GPU-based BWP algorithm was hardware-agnostic, meaning that it was not specifically tailored for the specific GPU hardware used for testing.

Other CUDA optimization techniques such as tiling, cache re-use, and exploiting warp characteristics were not used due to the fact that such approaches often impose restrictions on the optimization algorithm's parameters, like the number of decision variables, population size, or types of allowed combinations. A generic parallelization, such as the one presented in this paper, is important since it can run on a variety of hardware configurations and adapt to different optimization scenarios while still providing a significant speedup boost. This approach strikes an appropriate balance between adaptability and performance.

Acknowledgment

The authors would like to thank Emiliano Gonçalves for kindly providing access to the GPU hardware used in the computational experiments.

References

[1] R. Singh, K. Gaurav, V. Pathak, P. Singh, and H. Chaudhary, "Best–Worst–Play (BWP): A metaphor-less optimization algorithm," *J. Phys. Conf. Ser.*, vol. 1455, p. 012007, 2019.

[2] R. 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.

[3] ——, "Rao algorithms: Three metaphor-less simple algorithms for solving optimization problems," *Int. J. Ind. Eng. Comput.*, vol. 11, no. 1, pp. 107–130, 2020.

[4] J. Moré, B. Garbow, and K. Hillstrom, "Testing unconstrained optimization software," *ACM Trans. Math. Softw.*, vol. 7, no. 1, pp. 17–41, 1981.

[5] A. Friedlander, M. Gomes-Ruggiero, D. Kozakevich, J. Martínez, and S. Santos, "Solving nonlinear systems of equations by means of quasi-Newton methods with a nonmonotone strategy," *Optim. Methods Softw.*, vol. 8, no. 1, pp. 25–51, 1997.

[6] C. Kelley, L. Qi, X. Tong, and H. Yin, "Finding a stable solution of a system of nonlinear equations," *J. Ind. Manag. Optim.*, vol. 7, no. 2, pp. 497–521, 2011.

[7] E. Bodon, A. Del Popolo, L. Lukšan, and E. Spedicato, "Numerical performance of ABS codes for systems of nonlinear equations," Universitá degli Studi di Bergamo, Bergamo, Italy, Technical Report DMSIA 01/2001, 2001.

[8] M. Ziani and F. Guyomarc'h, "An autoadaptative limited memory Broyden's method to solve systems of nonlinear equations," *Appl. Math. Comput.*, vol. 205, no. 1, pp. 202–211, 2008.