GPU-Based Acceleration of Metaphor-less Algorithms and Their Application for Solving Large Systems of Nonlinear Equations

Bruno Silva and Luiz Guerreiro Lopes *

Posted Date: 1 May 2024

doi: 10.20944/preprints202405.0051.v1

Keywords: metaheuristic optimization; Jaya algorithm; Jaya variants; Rao algorithms; BWP algorithm; MaGI algorithm; parallel GPU algorithms; nonlinear equation systems

Article

GPU-Based Acceleration of Metaphor-Less Algorithms and Their Application for Solving Large Systems of Nonlinear Equations

Bruno Silva [1,2] and Luiz Guerreiro Lopes [3,*]

[1] Doctoral Program in Informatics Engineering, University of Madeira, Funchal, Madeira Is., Portugal
[2] Regional Secretariat for Education, Science and Technology, Regional Government of Madeira, Funchal, Portugal; bruno.silva@madeira.gov.pt
[3] Faculty of Exact Sciences and Engineering, University of Madeira, 9020-105 Funchal, Madeira Is., Portugal
* Correspondence: lopes@uma.pt; Tel.: +351-291-705-200

Abstract: Traditional population-based metaheuristic algorithms are effective in solving complex real-world problems but require careful strategy selection and parameter tuning. Parameter-less population-based optimization algorithms have gained importance due to their simplicity and efficiency. However, research on their applicability for solving large systems of nonlinear equations is still incipient. In this paper, a systematic review and detailed description of metaphor-less optimization algorithms is made, including the Jaya algorithm, the main Jaya variants for continuous optimization problems, with a focus on the Enhanced Jaya (EJAYA) algorithm, the three Rao algorithms, the best-worst-play (BWP) algorithm, and the very recent max-min-greedy-interaction (MaGI) algorithm. This paper also discusses a few previous parallelizations of the Jaya optimization algorithm, as well as recent GPU-based massively parallel implementations of the Jaya, Enhanced Jaya (EJAYA), Rao, and BWP algorithms developed by the authors. In addition, a novel GPU-accelerated version of the MaGI algorithm is proposed. The GPU-accelerated versions of the metaphor-less algorithms developed by the authors were implemented using the Julia programming language and tested primarily on GeForce RTX 3090 and NVIDIA A100 GPUs, as well as on Tesla V100S and Tesla T4 GPUs, using a set of difficult, large-scale nonlinear equation system problems. The computational experiments carried out produced quite significant speedups, which highlights the efficiency of the GPU-based versions of the metaphor-less algorithms developed for solving large-scale systems of nonlinear equations

Keywords: metaheuristic optimization; Jaya algorithm; Jaya variants; Rao algorithms; BWP algorithm; MaGI algorithm; parallel GPU algorithms; nonlinear equation systems

MSC: 65H10; 68W10; 90C59

1. Introduction

Nonlinear equations and systems of nonlinear equations (SNLEs) are ubiquitous in simulations of physical phenomena [1] and play a crucial role in pure and applied sciences [2]. These types of systems are considered among the most challenging problems to solve in numerical mathematics [3,4], and their difficulty increases significantly as the number of equations in the system grows.

The well-known Newton's method and its variants [5] are among the most common iterative numerical techniques used to solve SNLEs. The effectiveness of these solvers is strongly dependent on the quality of the initial approximations used [6,7], and convergence may not be guaranteed. Additionally, they can be very computationally demanding, especially for large and complex systems. Moreover, there is no sufficiently efficient and robust general numerical method for solving SNLEs [8]. Hence, it is crucial to explore alternative approaches to tackle the challenge of obtaining accurate and efficient approximations to the solutions of SNLEs.

In recent years, there has been growing interest in population-based metaheuristic algorithms for their ability to tackle complex and challenging numerical problems, such as SNLEs. These algorithms belong to a broader category of computational intelligence techniques, offering a versatile and robust

 doi:10.20944/preprints202405.0051.v1

approach to optimization [9]. However, although they are capable of addressing large-scale and high-dimensional problems, providing near-optimal solutions within a reasonable timeframe, it is important to note that they do not guarantee optimal (i.e., exact) solutions.

Population-based metaheuristic algorithms, typically drawing inspiration from natural evolution and social behavior, encompass various techniques such as particle swarm optimization, genetic algorithms, evolutionary strategies, evolutionary programming, genetic programming, artificial immune systems, and differential evolution. These algorithms utilize principles observed in nature to guide their search for optimal solutions. Numerous modifications have been proposed to make such algorithms more efficient, and their inherent characteristics make them natural candidates for parallelization. However, their effective application requires a careful selection of the appropriate strategy and the tuning of specific parameters, which is neither simple nor obvious.

Metaphor-less optimization algorithms, in contrast, operate independently of metaphors or analogies and are considered parameter-less since they do not require any parameter tuning. Originating from the pioneering work of R. V. Rao [10], these methods have gained increasing importance due to their relative simplicity and efficiency in solving problems across various domains with varying levels of complexity. These algorithms are often more abstract and rely on mathematical principles or specific optimization strategies rather than mimicking real-world phenomena.

Given the complexities associated with solving large-scale SNLEs, high-performance computing (HPC) techniques, such as the use of heterogeneous architectures, become crucial to ensuring access to adequate computational resources and achieving accurate and timely solutions. The deployment of parallel algorithms on Graphics Processing Units (GPUs) used as general-purpose computing devices has become a fundamental aspect of computational efficiency. These hardware units are equipped with a multitude of cores, allowing for the execution of numerous parallel operations simultaneously, thus surpassing the capabilities of traditional Central Processing Units (CPUs).

The integration of parallel algorithms into GPUs represents a significant advancement in HPC, providing an effective way to achieve scalability and efficiency at a relatively affordable price and effectively accelerating computations across diverse domains, including scientific simulations, artificial intelligence, financial modeling, and data analytics. The GPU-based parallel acceleration of metaheuristics allows for the efficient resolution of large, complex optimization problems and facilitates the exploration of vast solution spaces that would otherwise be impractical with traditional sequential computation due to the computational costs.

In this paper, a systematic review and detailed description of metaphor-less algorithms is made, including the Jaya optimization algorithm [10], the Enhanced Jaya (EJAYA) algorithm [11], a variant of Jaya that has recently been shown by the authors to be quite effective in solving nonlinear equation systems [12], the three Rao algorithms [13], the best-worst-play (BWP) algorithm [14], and the very recent max-min-greedy-interaction (MaGI) algorithm [15]. This paper also briefly discusses a few previous parallelizations of the Jaya optimization algorithm, as well as very recent and efficient GPU-based massively parallel implementations of the Jaya, Enhanced Jaya (EJAYA), Rao, and BWP algorithms developed by the authors. In addition, a novel GPU-accelerated version of the MaGI algorithm is proposed. The GPU-accelerated versions of the metaphor-less algorithms developed by the authors and tested on a suite of GPU devices using a set of difficult, large-scale SNLEs.

SNLEs must be reformulated into optimization problems for solving them using metaheuristics. Considering a SNLE represented generically in vector form as $f(\boldsymbol{x}) = \mathbf{0}$, with components $f_i(x_1, \ldots, x_n) = 0,\ \ i = 1, \ldots, n$, this system of n nonlinear equations with n unknowns can be transformed into an n-dimensional nonlinear minimization problem. Transforming SNLEs into optimization problems involves defining an objective function to minimize. In this paper, the objective function is defined as the sum of the absolute values of the residuals, $\sum_{i=1}^{n} |f_i(x_1, \ldots, x_n)|$.

In population-based optimization algorithms, the generation of the initial population commonly involves the random or strategic generation of a group of potential candidates to serve as the starting point for the optimization process. As a standard practice, the default initialization approach for the

 doi:10.20944/preprints202405.0051.v1

majority of metaheuristic algorithms involves the use of random number generation, or the Monte Carlo method [16], in which individuals within the initial population are generated randomly within predefined bounds or limitations. This approach does not guarantee an even distribution of candidate solutions across the search space [17] and can influence the performance of the algorithms. However, tests have shown that this behavior is contingent on both the test problems and the algorithm under consideration [16], indicating that certain combinations of problem domains and algorithms are almost impervious to this scenario. Considering the prevalent use of randomly generated starting populations and the scarce number of experiments employing alternative initialization procedures, a stochastic population initialization approach was also adopted.

The organization of this paper is as follows: The theoretical aspects and implementation details of the aforementioned metaphor-less algorithms are provided in Section 2. Principles behind GPU programming and details about the GPU-based implementation of each algorithm, including aspects about the different parallelization strategies employed, are presented in Section 2. All aspects of the experimental setup and methodology used to evaluate the performance of GPU-based algorithms are detailed in Section 4, including details about the hardware and benchmark functions used. The computational experiment findings and discussion are presented in Section 5. Finally, the conclusions drawn from the key findings are presented in Section 6.

2. Metaphor-less Optimization Algorithms

2.1. Jaya Algorithm

The Jaya optimization algorithm was introduced by R. V. Rao in 2016 with the purpose of optimizing both unconstrained and constrained problems. Jaya (a Sanskrit term meaning 'victory') stands out as a metaphor-less algorithm, representing an unprecedented and unique feature in the field of population-based metaheuristic algorithms. It is regarded as a parameter-less algorithm because it relies solely on standard control specifications, such as the maximum number of iterations and population size, and does not require any algorithm-specific control parameters.

Owing to its simplicity and efficiency, the Jaya algorithm has garnered growing interest and has emerged as a viable alternative to other widely recognized metaheuristic algorithms like Particle Swarm Optimization (PSO) [18] in various scientific and engineering disciplines (see, e.g., [19,20]).

The process of updating the population in Jaya in order to optimize a given objective function entails adjusting the current solutions by directing the search toward more favorable regions within the solution space while simultaneously moving away from potential solutions with worse fitness.

Considering the parameters associated with the given problem, encompassing the number of decision variables $numVar$, a variable index v ranging from 1 to $numVar$, the population size $popSize$, a population index p varying from 1 to $popSize$, and the iteration index i ranging from 1 to $maxIter$, the value of the v-th variable for the p-th population candidate at the i-th iteration is represented as $X_{p,v,i}$, and the updated value $X^{new}_{p,v,i}$ is determined based on the following equation:

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

where $X_{best,v,i}$ and $X_{worst,v,i}$ denote the population candidates with the best and worst fitness values, while $r_{1,v,i}$ and $r_{2,v,i}$ represent uniformly distributed random numbers within $[0, 1]$.

The pseudocode for the Jaya algorithm is delineated in Algorithm 1. The general framework of the computational procedure of this algorithm can be succinctly expressed with the following steps: the initialization, where the initial set of potential solutions is generated; the main loop, where the refinement of the candidate solutions is performed using Jaya equation (1), accompanied by the selection of the best candidate solution between the current and updated candidates until the termination criteria is satisfied; and the reporting of the best solution identified at the end of the algorithm.

 doi:10.20944/preprints202405.0051.v1

Algorithm 1 Jaya pseudocode

```
/* Initialization */
Initialize numVar, popSize and maxIter;
X ← GENERATE_INITIAL_POPULATION();
Evaluate fitness value f(X);
i ← 1;
/* Main loop */
while i ≤ maxIter do
    Determine X_{best,v,i} and X_{worst,v,i};
    for p ← 1, popSize do
        for v ← 1, numVar do
            Update population X^{new}_{p,v,i} by Eq. (1);
            CHECK_BOUNDARY_INTEGRITY(X^{new}_{p,v,i});
        end for
        Evaluate fitness value f(X^{new}_{p,v,i});
        GREEDY_SELECTION(X_{p,v,i}, X^{new}_{p,v,i});
    end for
    i ← i + 1;
end while
Output the best solution found and terminate;
```

The initial population of candidate solutions is arbitrarily positioned within the specific constraints of the optimization problem, employing random numbers uniformly distributed in $[0, 1]$. A detailed description of the procedure used is presented in Algorithm 2. The lower bound (LB) and upper bound (UB) parameters determine the size of the search space and the boundaries for each decision variable. These parameters are specific to the characteristics of the optimization problem under consideration.

Algorithm 2 Initial population generation

```
function GENERATE_INITIAL_POPULATION( )
    for p ← 1, popSize do
        for v ← 1, numVar do
            X_{p,v} ← LB_v + rand() × (UB_v − LB_v);
        end for
    end for
end function
```

To maintain the feasibility of solutions and prevent candidates from going outside the search space, the range of decision variable values of every newly generated candidate solution (X_{new}) has to be checked and corrected to conform to the problem-specific constraints.

This is done by checking if there are decision variables that fall outside the lower or upper bounds and bringing them back to the search space boundaries, in accordance with Algorithm 3.

Algorithm 3 Boundary integrity control

```
function CHECK_BOUNDARY_INTEGRITY(X^{new}_{p,v})
    if X^{new}_{p,v} > UP_v then
        X^{new}_{p,v} ← UP_v;
    end if
    if X^{new}_{p,v} < LB_v then
        X^{new}_{p,v} ← LB_v;
    end if
end function
```

A common characteristic of this class of metaheuristics is the use of a greedy selection algorithm to expedite the convergence speed of the optimization process. The procedure, outlined in Algorithm 4, is employed locally to choose the best candidate (i.e., the candidate solution with the best fitness

 doi:10.20944/preprints202405.0051.v1

value) between the newly generated candidate and the current candidate solution under consideration. The evaluation of the best fitness value is tied to the optimization goal, which may involve either minimizing or maximizing an objective function.

Algorithm 4 Greedy selection

```
function GREEDY_SELECTION(X_{p,v}, X^{new}_{p,v})
    if f(X^{new}_{p,v}) is better than f(X_{p,v}) then
        X_{p,v} ← X^{new}_{p,v};
        f(X_{p,v}) ← f(X^{new}_{p,v});
    else
        Keep X_{p,v} and f(X_{p,v}) values;
    end if
end function
```

2.2. Enhanced Jaya Algorithm

The Enhanced Jaya (EJAYA) algorithm [11] is a metaheuristic build upon the Jaya algorithm to address some of its limitations, namely the limited focus on the current best and worst solutions to guide the search process that might lead to premature convergence. This issue is tackled in EJAYA through the introduction of a more comprehensive exploration strategy capable of leveraging different search methods and incorporate additional information about the population to effectively exploit and explore possible solutions. This strategy aims to safeguard population diversity, navigate away from local optima, and facilitate further enhancement and refinement of candidate solutions.

Similarly to the original Jaya algorithm, EJAYA incorporates information regarding the best and worst solutions to explore the search space. However, it distinguishes itself by utilizing supplementary parameters, including the mean solution and historical solutions, to harmonize its Local Exploitation Strategy (LES) and Global Exploration Strategy (GES).

The LES phase involves the calculation of two elements, the upper and lower local attract points. These attract points are employed to direct the population towards potentially superior solutions and concurrently mitigating the risk of premature convergence.

The upper local attract point (PU) serves to characterize the solution positioned between the current best solution (X_{best}) and the current mean solution, determined in the following way:

$$PU = r_3 \times X_{best} + (1 - r_3) \times M, \tag{2}$$

where r_3 represents a random number that follows a uniform distribution over the interval from 0 to 1, and M signifies the current population mean defined as:

$$M = \frac{1}{popSize} \sum_{p=1}^{popSize} X_p. \tag{3}$$

The lower local attract point (PL) is determined similarly to the upper local attract point, but instead it represents the solution in between the present solution with the worst fitness (X_{worst}) and the mean solution, through the following formulation:

$$PL = r_4 \times X_{worst} + (1 - r_4) \times M, \tag{4}$$

where r_4 signifies a random number following a uniform distribution over the interval $[0, 1]$.

By leveraging both the PU and PL attract points, the LES phase of EJAYA can be described as follows:

$$X^{new}_{p,v,i} = X_{p,v,i} + r_{5,v,i}(PU_i - X_{p,v,i}) - r_{6,v,i}(PL_i - X_{p,v,i}), \tag{5}$$

where $r_{5,v,i}$ and $r_{6,v,i}$ indicate randomly generated numbers uniformly distributed over the interval $[0, 1]$.

 doi:10.20944/preprints202405.0051.v1

In an effort to more comprehensively explore the global search space, EJAYA utilizes a GES phase that is based on the principles of the Backtracking Search Optimization Algorithm [21]. This is accomplished by alternatively leveraging information from either the current population ($X_{p,v}$) or a historical population ($X^{old}_{p,v}$).

In the first iteration (i.e., at the start of the algorithm), the historical population $X^{old}_{p,v}$ coincides with $X_{p,v}$; however, in the following iterations, it is determined by a switching probability defined in the following way:

$$X^{old}_{p,v} = \begin{cases} X_{p,v} & \text{if } P_{switch} \leq 0.5 \\ X^{old}_{p,v} & \text{otherwise,} \end{cases} \tag{6}$$

with P_{switch} denoting a random number uniformly distributed between 0 and 1.

Subsequently, the historical population $X^{old}_{p,v}$ undergoes the following procedure:

$$X^{old}_{p,v} = permuting\left(X^{old}_{p,v}\right), \tag{7}$$

where *permuting* represents a shuffling function that reorders each individual (i.e., candidate solution) within $X^{old}_{p,v}$ randomly.

Finally, the main operational procedure for the GES phase is expressed as follows:

$$X^{new}_{p,v,i} = X_{p,v,i} + k \times \left(X^{old}_{p,v,i} - X_{p,v,i}\right), \tag{8}$$

with k representing a randomly sampled value with a standard normal distribution spanning from 0 to 1.

Given the pivotal and complementary roles played by both the LES and GES phases in the optimization process, EJAYA ensures a balanced utilization of these search strategies. This equilibrium is maintained through an equivalent switching probability, as expressed by the following formulation:

$$\text{Search strategy} = \begin{cases} LES & \text{if } P_{select} > 0.5 \\ GES & \text{otherwise,} \end{cases} \tag{9}$$

where P_{select} represents a randomly generated number following a uniform distribution in the range of 0 to 1.

Algorithm 5 presents pseudocode providing a comprehensive overview of the distinct procedural steps constituting EJAYA.

 doi:10.20944/preprints202405.0051.v1

Algorithm 5 EJAYA pseudocode

```
/* Initialization */
Initialize numVar, popSize and maxIter;
X ← GENERATE_INITIAL_POPULATION();
Evaluate fitness value f(X);
X^old ← X;
i ← 1;
/* Main loop */
while i ≤ maxIter do
    Determine X_best,v,i and X_worst,v,i;
    Determine PU_i by Eq. (2);
    Determine PL_i by Eq. (4);
    if P_switch ≤ 0.5 then
        X_i^old ← X_i;
    end if
    Permute X_i^old by Eq. (7);
    for p ← 1, popSize do
        if P_select > 0.5 then                          ▷ Local exploitation strategy
            for v ← 1, numVar do
                Update population X_p,v,i^new by Eq. (5);
                CHECK_BOUNDARY_INTEGRITY(X_p,v,i^new);
            end for
        else                                            ▷ Global exploration strategy
            for v ← 1, numVar do
                Update population X_p,v,i^new by Eq. (8);
                CHECK_BOUNDARY_INTEGRITY(X_p,v,i^new);
            end for
        end if
        Evaluate fitness value f(X_p,v,i^new);
        GREEDY_SELECTION(X_p,v,i, X_p,v,i^new);
    end for
    i ← i + 1;
end while
Output the best solution found and terminate;
```

2.3. Rao Optimization Algorithms

The Rao optimization algorithms were proposed in 2020 by R. V. Rao [13], the same author of the Jaya algorithm, and are composed by three different optimization algorithms, denoted as Rao-1, Rao-2, and Rao-3.

These algorithms represent a concerted endeavor to advance the field of optimization by devising simple and efficient metaphor-less methodologies. A distinctive feature of these algorithms is their deliberate avoidance of algorithm-specific control parameters. This underscores an emphasis on common parameters such as population size and the number of iterations, guided by a simplicity-driven perspective. The motivation behind these developments is grounded in a broader strategy to fostering optimization techniques that eschew reliance on metaphors while simultaneously maintaining algorithm robustness.

The three Rao algorithms share the common approach of incorporating information about the current best and worst solutions and random factors to update and direct the population towards an optimal solution but each algorithm uses different strategies.

In the context of the Rao-1 algorithm, the updated value ($X^{new}_{p,v,i}$) is determined by the following equation:

$$X^{new}_{p,v,i} = X_{p,v,i} + r_{7,v,i}(X_{best,v,i} - X_{worst,v,i}), \tag{10}$$

where $r_{7,v,i}$ is a randomly generated number uniformly distributed over the interval $[0, 1]$.

The Rao-2 and Rao-3 algorithm are extended versions of Rao-1 with an additional segment in their update equations that introduce a second random factor and random interactions between

 doi:10.20944/preprints202405.0051.v1

candidates based on fitness values. The additional equation segment of the Rao-2 and Rao-3 equations are contingent upon whether the fitness value of the current candidate solution, denoted as $f(X_{p,v,i})$, is better than the fitness value of the randomly chosen solution, $f(X_{t,v,i})$. The Rao-2 (Eq. 11) and Rao-3 (Eq. 12) main functions are described below:

$$X^{new}_{p,v,i} = X_{p,v,i} + r_{8,v,i}(X_{best,v,i} - X_{worst,v,i}) + \begin{cases} r_{9,v,i}(|X_{p,v,i}| - |X_{t,v,i}|) & \text{if } f(X_{p,v,i}) \text{ better than } f(X_{t,v,i}) \\ r_{9,v,i}(|X_{t,v,i}| - |X_{p,v,i}|) & \text{otherwise,} \end{cases} \tag{11}$$

$$X^{new}_{p,v,i} = X_{p,v,i} + r_{10,v,i}(X_{best,v,i} - X_{worst,v,i}) + \begin{cases} r_{11,v,i}(|X_{p,v,i}| - X_{t,v,i}) & \text{if } f(X_{p,v,i}) \text{ better than } f(X_{t,v,i}) \\ r_{11,v,i}(|X_{t,v,i}| - X_{p,v,i}) & \text{otherwise.} \end{cases} \tag{12}$$

In this context, $r_{8,v,i}$, $r_{9,v,i}$, $r_{10,v,i}$ and $r_{11,v,i}$ are random variables uniformly distributed in the range of 0 to 1, and $X_{t,v,i}$ denotes a randomly selected candidate solution from the population (i.e., $t \in [1, popSize]$) that is distinct from $X_{p,v,i}$ (i.e., $t \neq p$).

An unified description of the pseudocode for the Rao-1, Rao-2, and Rao-3 algorithms is presented in Algorithm 6.

Algorithm 6 Rao-1, Rao-2, and Rao-3 pseudocode

```
/* Initialization */
Initialize numVar, popSize and maxIter;
X ← GENERATE_INITIAL_POPULATION();
Evaluate fitness value f(X);
i ← 1;
/* Main loop */
while i ≤ maxIter do
    Determine X_{best,v,i} and X_{worst,v,i};
    for p ← 1, popSize do
        if algorithm = 'Rao-2' or 'Rao-3' then
            t ← Random candidate solution different than p;
        end if
        for v ← 1, numVar do
            if algorithm = 'Rao-1' then
                Update population X^{new}_{p,v,i} by Eq. (10);
            else if algorithm = 'Rao-2' then
                Update population X^{new}_{p,v,i} by Eq. (11);
            else if algorithm = 'Rao-3' then
                Update population X^{new}_{p,v,i} by Eq. (12);
            end if
            CHECK_BOUNDARY_INTEGRITY(X^{new}_{p,v,i});
        end for
        Evaluate fitness value f(X^{new}_{p,v,i});
        GREEDY_SELECTION(X_{p,v,i}, X^{new}_{p,v,i});
    end for
    i ← i + 1;
end while
Output the best solution found and terminate;
```

2.4. Best-Worst-Play Algorithm

The Best-Worst-Play (BWP) algorithm [14] is a metaphor-less optimization approach that combines elements from two algorithms, Jaya [10] and Rao-1 [13]. The algorithm aims to strike a balance between

 doi:10.20944/preprints202405.0051.v1

exploration and exploitation in solving unconstrained and constrained optimization problems by employing two operators inspired by Jaya and Rao-1 that are executed consecutively.

Through a process of hybridization, the BWP algorithm preserves an approach analogous to that employed by Jaya and Rao-1, indicating its capability to progressively improve the quality of the population over time, ultimately converging toward a near-optimal solution by leveraging information regarding both the best and worst candidate solutions. In this manner, BWP has managed to remain a straightforward and parameter-less algorithm.

As mentioned before, the update equation for the BWP algorithm incorporates both the Jaya and Rao-1 heuristics. In the case of Jaya, it employs the same unaltered method (see Eq. 1), while the Rao-1 method undergoes a modification in the $X_{worst,v,i}$ parameter, as shown below:

$$X^{new}_{p,v,i} = X_{p,v,i} + r_{12,v,i}(X_{best,v,i} - |X_{worst,v,i}|), \tag{13}$$

with $r_{12,v,i}$ being a random variable with a uniform distribution from 0 to 1.

These two methods are applied one after the other to all candidate solutions across all iterations, as illustrated in Algorithm 7.

Algorithm 7 BWP pseudocode

```
/* Initialization */
Initialize numVars, popSize and maxIters;
X ← GENERATE_INITIAL_POPULATION();
Evaluate fitness value f(X);
i ← 1;
/* Main loop */
while i ≤ maxIters do
    /* Leverage the Jaya heuristic */
    Determine X_{best,v,i} and X_{worst,v,i};
    for p ← 1, popSize do
        for v ← 1, numVars do
            Update population X^{new}_{p,v,i} by Eq. (1);
            CHECK_BOUNDARY_INTEGRITY(X^{new}_{p,v,i});
        end for
        Evaluate fitness value f(X^{new}_{p,v,i});
        GREEDY_SELECTION(X_{p,v,i}, X^{new}_{p,v,i});
    end for
    /* Leverage the Rao-1 based heuristic */
    Determine X_{best,v,i} and X_{worst,v,i};
    for p ← 1, popSize do
        for v ← 1, numVars do
            Update population X^{new}_{p,v,i} by Eq. (13);
            CHECK_BOUNDARY_INTEGRITY(X^{new}_{p,v,i});
        end for
        Evaluate fitness value f(X^{new}_{p,v,i});
        GREEDY_SELECTION(X_{p,v,i}, X^{new}_{p,v,i});
    end for
    i ← i + 1;
end while
Output the best solution found and terminate;
```

2.5. Max–Min-Greedy-Interaction Algorithm

The Max–Min-Greedy-Interaction (MaGI) algorithm is a metaphor-less optimization approach recently published by R. Singh et al. [15] that demonstrated effectiveness in mechanism synthesis and exhibited promise for diverse optimization tasks.

The algorithm utilizes maximum and minimum values within the group of candidates, essentially representing the best and worst candidates, for updating the population. It incorporates operators

 doi:10.20944/preprints202405.0051.v1

inspired by the Jaya and Rao-2 algorithms, in addition to performing greedy interactions. This approach appears to be influenced by the BWP methodology, as both heuristics are executed sequentially throughout the iteration process. The Jaya main function (see Eq. 1) remains unmodified, whereas the Rao-2 method undergoes adjustments in several parameters, as indicated by the following equation:

$$X^{new}_{p,v,i} = X_{p,v,i} + r_{13,v,i}(X_{best,v,i} - |X_{worst,v,i}|) + \begin{cases} r_{14,v,i}(X_{p,v,i} - X_{t,v,i}) & \text{if } f(X_{p,v,i}) \text{ better than } f(X_{t,v,i}) \\ r_{14,v,i}(X_{t,v,i} - X_{p,v,i}) & \text{otherwise} \end{cases} \tag{14}$$

For this condition, $r_{13,v,i}$ and $r_{14,v,i}$ are chosen randomly from a uniform distribution spanning from 0 to 1.

The implementation outline, detailed in Algorithm 8, provides an in-depth explanation of the step-by-step procedures inherent in the MaGI methodology.

Algorithm 8 MaGI pseudocode

```
/* Initialization */
Initialize numVars, popSize and maxIters;
X ← GENERATE_INITIAL_POPULATION();
Evaluate fitness value f(X);
i ← 1;
/* Main loop */
while i ≤ maxIters do
    /* Leverage the Jaya heuristic */
    Determine X_{best,v,i} and X_{worst,v,i};
    for p ← 1, popSize do
        for v ← 1, numVars do
            Update population X^{new}_{p,v,i} by Eq. (1);
            CHECK_BOUNDARY_INTEGRITY(X^{new}_{p,v,i});
        end for
        Evaluate fitness value f(X^{new}_{p,v,i});
        GREEDY_SELECTION(X_{p,v,i}, X^{new}_{p,v,i});
    end for
    /* Leverage the Rao-2 based heuristic */
    Determine X_{best,v,i} and X_{worst,v,i};
    for p ← 1, popSize do
        for v ← 1, numVars do
            Update population X^{new}_{p,v,i} by Eq. (14);
            CHECK_BOUNDARY_INTEGRITY(X^{new}_{p,v,i});
        end for
        Evaluate fitness value f(X^{new}_{p,v,i});
        GREEDY_SELECTION(X_{p,v,i}, X^{new}_{p,v,i});
    end for
    i ← i + 1;
end while
Output the best solution found and terminate;
```

3. GPU Parallelization of Metaphor-Less Optimization Algorithms

3.1. Principles of CUDA Programming

The graphics processing unit (GPU) has become an essential component of modern computers. Initially designed for rendering graphics, this hardware has evolved to manage simultaneously a multitude of graphical elements, such as pixels or vertices, for which a large parallel processing capacity is required.

The Compute Unified Device Architecture (CUDA) [22] is a computing platform developed by NVIDIA that harnesses the power of GPUs for general-purpose parallel computing, making it

 doi:10.20944/preprints202405.0051.v1

possible to take advantage of the massive parallelism offered by the GPU for distinct computing tasks. The CUDA software stack acts as a framework that abstracts the inherent complexities of GPU programming. This streamlines communication and coordination between the central processing unit (CPU) and GPU, providing an interface to efficiently utilize GPU resources in order to accelerate specific functions within programs. The functions designed to execute highly parallelized tasks on the GPU are referred to as kernels.

CUDA functions as a heterogeneous computing framework, consequently dictating a specific execution pattern within kernels The CPU functions as the host, responsible for initiating kernel launches and overseeing the control flow of the program. On the other hand, the kernels are executed in parallel by numerous threads on the GPU, designated as the device. In essence, the host performs tasks that are better suited for sequential execution, while the GPU handles the computationally intensive parts of the program in parallel. This relationship is illustrated in Figure 1.

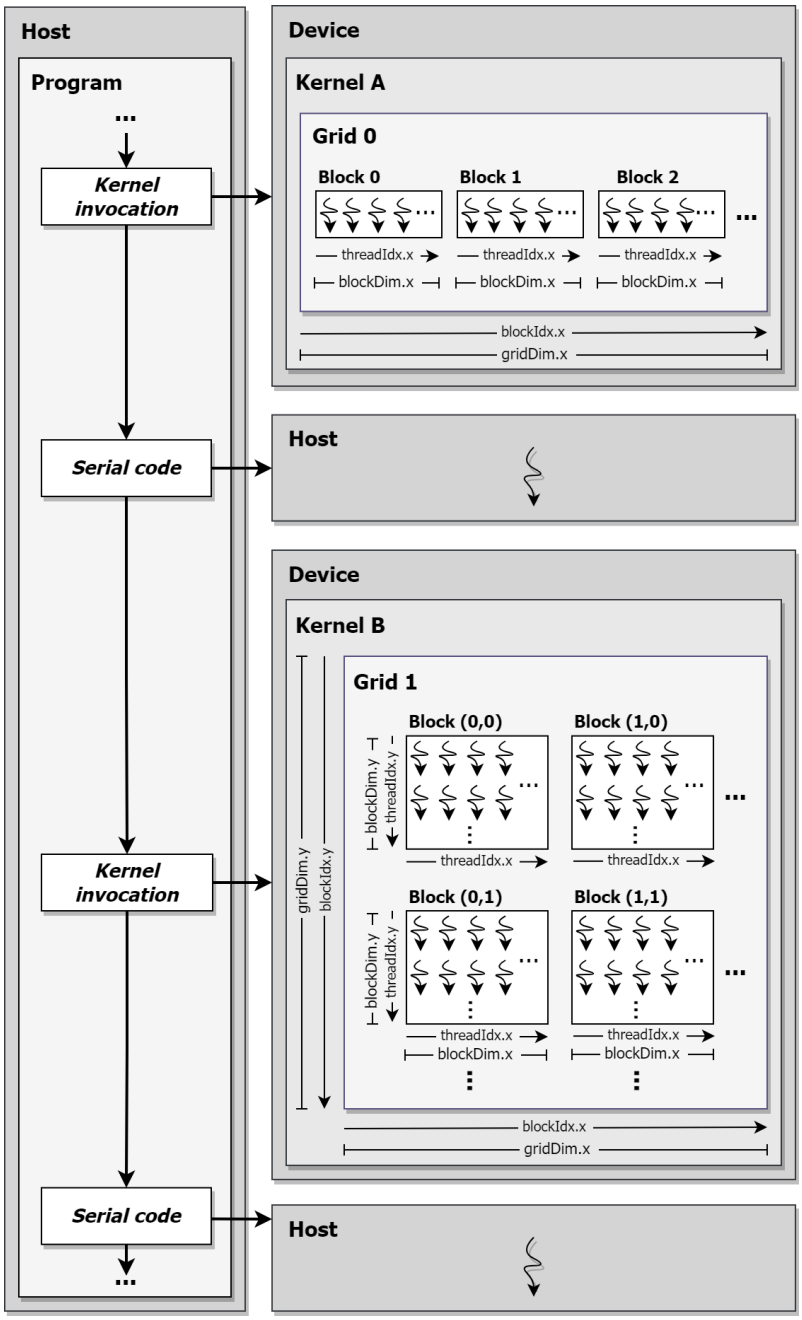

Figure 1. Heterogeneous computing architecture with 1D and 2D thread hierarchy in CUDA kernels.

 doi:10.20944/preprints202405.0051.v1

The computational resources of the GPU are organized into a grid to execute multiple kernels in parallel. This grid acts as an abstraction layer, structuring the available computational elements into blocks, each containing a set number of threads. This thread hierarchy is configurable into grids with one, two, or three dimensions, tailored to accommodate the parallelization needs of a specific kernel. The configuration parameters within the thread hierarchy play a pivotal role in determining GPU hardware occupancy, making them a critical aspect of optimizing resource utilization.

In Figure 1, *Kernel A* showcases a one-dimensional thread-block arrangement, while *Kernel B* displays a two-dimensional configuration. The illustration also includes the CUDA built-in variables for accessing the grid and block dimensions (*gridDim* and *blockDim*), block and thread indices (*blockIdx* and *threadIdx*), and their respective coordinates (x and y). These variables also facilitate the unique identification of individual threads within a kernel through a process known as thread indexing.

The discrete nature of the host and device hardware involves separate memory spaces, preventing direct data access between them. This necessitates explicit data transfers, resulting in substantial computational overheads and performance degradation. Consequently, meticulous optimization of memory access patterns and a reduction in the frequency of data transfer are crucial to mitigating performance bottlenecks [23].

CUDA also provides access to diverse memory types within the GPU memory hierarchy, which is an important consideration for performance optimization. The memory types encompass global, shared, and local memories, each serving distinct purposes. The global memory is the largest memory segment on the device but is also the slowest. It remains persistent throughout kernel execution, serves as the interface for data transfers between the host and device, and is accessible by all threads in the GPU, thus enabling global data sharing. The shared memory, though much smaller, offers greater speed but is exclusive to threads within a block, enabling efficient data exchange among them. Lastly, local memory is the smallest in size and is private to individual threads, allowing them to store their specific local variables and support thread local execution.

3.2. Methodology Used for GPU-Based Parallelization

The primary step in parallelizing any algorithm requires an extensive assessment of its architecture, dependencies, data structure, etc., to devise a viable parallelization strategy. While some aspects may not be conducive to parallel execution, others, albeit parallelizable, may pose implementation challenges.

Several studies focusing on parallel and distributed metaheuristic algorithms have employed implementation strategies based on decomposing the optimization problem into smaller, independent subproblems (see e.g., [**?**]). These subproblems can then be distributed across multiple parallel computation units, sometimes running several sequential instances of metaheuristic algorithms concurrently. An alternative method involves running specific phases of the algorithm in parallel, while others are executed sequentially (see e.g., [**?**]). However, these approaches often prove less efficient due to the loss of computational resources attributed to overheads related to data transfers, synchronization, and other communication mechanisms.

While these parallel strategies can notably enhance computational performance compared to their original sequential counterparts, they are suboptimal for GPU computing due to their tendency to underutilize the available computational resources [24]. Furthermore, specific parallelization strategies can result in modifications in the behavior of the parallel algorithm in contrast to its original version, leading to variations in convergence characteristics and the quality of the obtained approximations.

As CUDA inherently adopts a heterogeneous approach, establishing an effective balance between the host and device is crucial. In the proposed GPU-based parallelization, the primary responsibilities of the host include initializing algorithm variables, invoking kernel functions, managing the primary loop, and reporting the best solution found. Consequently, all intensive computations are exclusively handled by the device.

 doi:10.20944/preprints202405.0051.v1

Given this perspective, the proposed methodology for GPU-based parallelization of metaphor-less optimization algorithms must ensure optimal utilization of the available GPU resources while still aligning with the fundamental operating principles of the original algorithm. This is essential to ensure a similar convergence behavior and quality of the obtained solutions in the parallelized version. Additionally, maintaining the same flexibility as the original algorithm is paramount, ensuring that the parallel algorithm can operate under virtually any optimization parameters.

To accomplish these objectives and handle large-scale optimization problems efficiently, the GPU-based parallelization design emphasizes the data structures of the optimization algorithms. In this approach, the GPU computational units are assigned based on the size and structure of the data segment being processed. Consequently, scalability is achieved when the number of threads allocated closely matches the size of the data computed. This leads to the alignment of both one-dimensional (1D) and two-dimensional (2D) arrays with the corresponding CUDA thread hierarchy, facilitating their processing by the respective kernels and addressing data locality issues.

Figure 2 illustrates the data-to-thread indexing of the largest data structure, a 2D array containing population data, to a 2D grid arrangement on the device. This configuration enables parallel computation across multiple blocks of threads. In this example, data corresponding to variable 6 from candidate 8 (using 0 as the starting index) is processed by the thread (3,2) within block (1,1). For the parallel processing of 1D array data, such as fitness values from each candidate solution, a 1D thread-block arrangement is utilized.

Optimizing the number of blocks and threads per block is critical for leveraging hardware resources efficiently. However, determining the optimal configuration is not straightforward; it varies based on algorithm characteristics, GPU architecture, memory needs, and workload specifics. Traditional methods for determining this configuration involve iterative experimentation, optimization, and profiling, which can be time-consuming.

In the proposed GPU-based parallelization approach, a dynamic and generalized method to automatically determine optimal GPU hardware occupancy was developed. This process involves a function that predicts the launch configurations for each kernel by leveraging CUDA runtime methods available in devices supporting CUDA 6.5 or higher, utilizing parameters such as data size, supported block size, and kernel memory requirements. To suit the algorithmic data being processed by optimization algorithms, two kernel launch configuration functions were created, one for 1D and another for 2D thread-block kernels. This approach ensures that kernels are adaptable to diverse GPU hardware configurations and are inherently structured for horizontal scalability through the utilization of additional available threads, all achieved without the need for modifications to the parallel implementation. Although automatic methods present the potential for achieving maximum GPU occupancy, they may not consistently produce the most efficient parameters tailored to a specific hardware setup.

In this study, the method for automatically determining optimal GPU hardware occupancy has undergone revision. The modification entails the incorporation of a parameter designed to constrain the suggested block size (i.e., the number of threads allocated for computation in each block used) to a multiple of the supported warp size by the GPU. The warp size, which denotes the minimum number of threads that can be concurrently executed in CUDA, is inherently dependent on the GPU architecture and can be automatically derived from the hardware parameters. Empirical tests have revealed that constraining the suggested block size to twice the warp size yields an additional average performance improvement of 19.8% compared to the preceding version of the automated launch configuration method when using the RTX 3090 GPU. This enhancement is primarily attributed to the facilitated coalesced memory access and optimized instruction dispatch achieved through the harmonization of block size.

While both shared memory and local memory exhibit faster access compared to global memory due to their proximity to processing units, their limited size and accessibility pose challenges when handling extensive data volumes. Therefore, the proposed GPU-based implementation prioritizes the

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 May 2024 doi:10.20944/preprints202405.0051.v1

use of larger global memory. This memory type offers substantial storage capacity, accommodating extensive datasets, variables, diverse data types, and complex data structures. Moreover, it aids in thread synchronization, ensuring that when a kernel concludes execution and the subsequent one initiates, all processed data synchronizes in the GPU global memory. In scenarios where quicker memory access is more efficient, this approach presents a drawback by potentially limiting computational task performance. Nevertheless, it fosters a versatile implementation in terms of problem size, population size of the optimization algorithm, and GPU hardware compatible with this parallelization method.

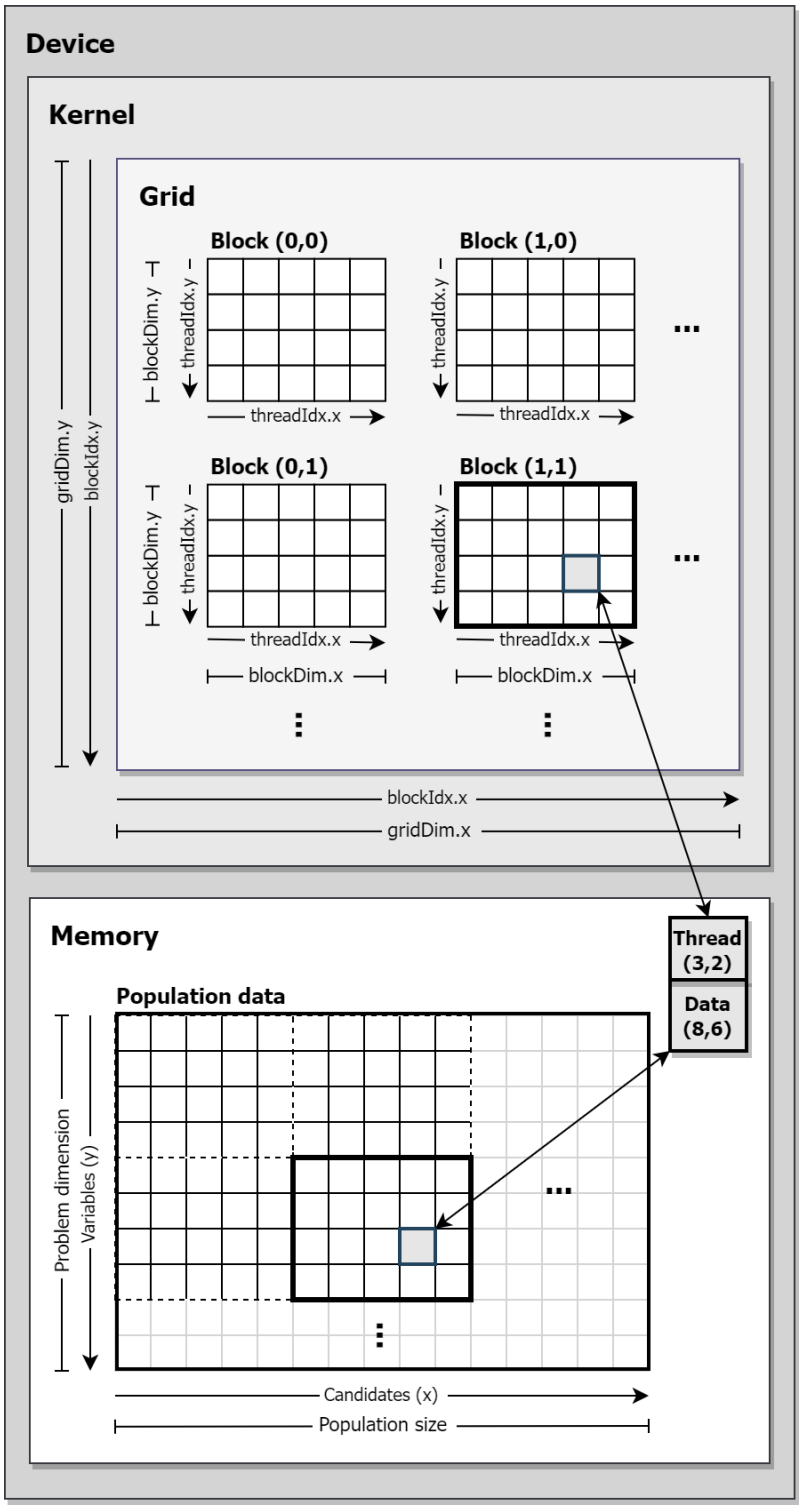

Figure 2. Population thread mapping, showing a 2D grid kernel and its relation with data memory.

 doi:10.20944/preprints202405.0051.v1

The GPU parallelization strategy presented aims to be versatile and adaptable. It is not only capable of utilizing resources across different GPU hardware but can also be seamlessly tailored to suit other metaheuristic optimization methods.

3.3. GPU-Based Acceleration of the Jaya Optimization Algorithm

GPU-Jaya was the first known parallelization of the Jaya algorithm on a GPU, introduced by Wang et al. in 2018 [25] to estimate the parameters of a Li-ion battery model. The parallelization strategy employed in this algorithm focuses on minimizing data transmission between the host and the GPU. It utilizes both global and shared memories to efficiently manage data location, and the algorithm incorporates a parallel reduction procedure to further optimize specific steps of the parallelization. Data is stored in a collection of one-dimensional arrays, leading to the establishment of a set of conversion and mapping mechanisms. This implies a potential computing overhead resulting from the use of data adaptations. The number of threads is fixed at 32, while the number of blocks is determined by the size of the population array ($popSize \times numVar$) divided by 32. This suggests that the population array may need to be a multiple of 32, imposing constraints on selectable test parameters.

The testing encompassed modest population sizes, ranging from 64 to 512. The problem involved six decision variables, with a maximum allowable number of iterations set at 20,000. With the fastest GPU used for testing, a Tesla K20c GPU with 2496 CUDA cores and 5 GB of VRAM, the GPU-Jaya achieved speedup factors ranging from 2.96 to 34.48, with an average of 13.17.

Jimeno-Morenilla and collaborators proposed in 2019 [26] another parallelization of the Jaya algorithm on the GPU. The parallel execution scheme in this implementation determines the number of blocks based on the number of algorithm runs, while the number of threads per block is established by the population size and the number of design variables. However, this approach has the disadvantage of leading to an inefficient use of GPU resources, as the hardware allocation is influenced by the number of algorithm runs rather than the dimensions of the test case. Moreover, it imposes notable constraints on the allowable range of test parameters, particularly the population size and the number of decision variables. These restrictions stem from the close correlation of these parameters with the hardware characteristics of the GPU used for testing, particularly the maximum number of threads per block.

The algorithm underwent evaluation on a GTX 970 GPU featuring 1664 CUDA cores and 2 GB of VRAM. It was utilized a set of 30 unconstrained functions with population sizes ranging from 8 to 256, encompassing 2 to 32 decision variables and constrained by a maximum iteration count of 30,000. The implementation achieved maximum speedup factors ranging from 18.4 to 189.6, with an average of 53.

A recent GPU parallelization of the Jaya algorithm proposed by the authors [27,28] makes use of the parallel execution strategy on GPUs detailed in the preceding section. The fundamental structure of this GPU-based version of Jaya is elucidated in Algorithm 9.

Algorithm 9 GPU-based parallel Jaya

1: /* *Initialization* */
2: Initialize $numVar$, $popSize$ and $maxIter$; ▷ *Host*
3: $X \leftarrow$ GENERATE_INITIAL_POPULATION_KERNEL();
4: EVALUATE_FITNESS_VALUES_KERNEL(X);
5: $i \leftarrow 1$; ▷ *Host*
6: /* *Main loop* */
7: **while** $i \leq maxIter$ **do** ▷ *Host*
8: Determine $X_{best,i}$ and $X_{worst,i}$;
9: $X_i^{new} \leftarrow$ JAYA-DRIVEN_POPULATION_UPDATE_KERNEL(X_i);
10: EVALUATE_FITNESS_VALUES_KERNEL(X_i^{new});
11: GREEDY_SELECTION_KERNEL(X_i, X_i^{new});
12: $i \leftarrow i + 1$; ▷ *Host*
13: **end while**
14: Output the best solution found and terminate; ▷ *Host and device*

 doi:10.20944/preprints202405.0051.v1

Upon initial observation, juxtaposing the GPU-based Jaya algorithm with its sequential counterpart (Algorithm 1) elucidates a notably analogous structure. They primarily adhere to identical execution procedures, diverging primarily due to the absence of nested loops iterating through the population elements (i.e., the candidate solutions) and decision variables. This aspect is a fundamental component of the established parallelization methodology, aiming to maintain the philosophical underpinnings of the original algorithm throughout its adaptation to GPU processing. As a result, the GPU-based version of Jaya retains the ability to generate identical optimization behavior to the original algorithm.

The provided procedure relies on a heterogeneous computational approach, where the computational involvement of the host is notably limited. As delineated in Algorithm 9, the primary tasks of the host entail initializing algorithmic parameters, invoking the various kernel functions, overseeing the algorithm's iterative steps (encapsulated in the main loop), and documenting the optimal solution identified. Consequently, the host manages the sequential aspects of the algorithm, whereas all optimization-related data resides and undergoes processing exclusively within the device. This operational paradigm ensures that no data is transferred between the host and device throughout the main computational process. In the final step of the algorithm, the device identifies the best solution discovered, which is then transferred to the host for output.

Given the aforementioned details, the generation of the initial population takes place directly within the device, following the procedure outlined in Algorithm 10.

Algorithm 10 Kernel for generating the initial population

```
1: function GENERATE_INITIAL_POPULATION_KERNEL( )
2:     /* Device code */
3:     Determine row using x dimension of blockDim, blockIdx, and threadIdx;
4:     Determine col using y dimension of blockDim, blockIdx, and threadIdx;
5:     if row ≤ popSize and col ≤ numVar then
6:         X[row, col] ← LB[col] + rand() × (UB[col] − LB[col]);
7:     end if
8: end function
```

In the context of the GPU-based algorithm, the conventional nested loops utilized for iterating through population data during processing have become obsolete due to concurrent parallel execution. Employing a two-dimensional thread block arrangement grid, the kernel specified in Algorithm 10 facilitates the parallel creation of the initial population. This methodology assigns an individual thread to generate each individual data element within the population (X), supported by a matrix data structure with dimensions $popSize \times numVar$. The kernel leverages the x and y coordinates (indices) from the block dimension (*blockDim*), block index (*blockIdx*), and thread index (*threadIdx*) to determine the specific row and column in the data matrix allocated for computation for assigning each data element individually to a separated thread for processing. This allocation strategy maximizes the efficiency of the parallel processing, allowing each thread to perform designated computations based on its unique position within the block and grid structure.

The same parallelization technique is applied across various algorithm phases encompassed within the main loop. Consequently, each stage of the computational process now runs concurrently on all elements within a given iteration. This paradigm shift means that X^{new} now signifies all updated candidate solutions within a given algorithmic iteration, rather than representing a singular updated candidate. Subsequently, all algorithmic steps have been adjusted accordingly, commencing with the population update kernel outlined in Algorithm 11.

 doi:10.20944/preprints202405.0051.v1

Algorithm 11 Kernel for population update (Jaya)

function JAYA-DRIVEN_POPULATION_UPDATE_KERNEL(X)
    /* *Device code* */
    Determine row using x dimension of $blockDim$, $blockIdx$, and $threadIdx$;
    Determine col using y dimension of $blockDim$, $blockIdx$, and $threadIdx$;
    if $row \leq popSize$ **and** $col \leq numVar$ **then**
        $X^{new}[row, col] = X[row, col] + rand() \times (X_{best}[col] - |X[row, col]|) - rand() \times (X_{worst}[col] - |X[row, col]|)$; ▷ *Eq. (1)*
        CHECK_BOUNDARY_INTEGRITY_FUNCTION(X^{new});
    end if
end function

Similar to the original algorithm, the maintenance of newly generated candidate solutions within the search space boundary remains a necessity. Within the GPU-based algorithm, this task is managed by the function delineated in Algorithm 12.

Algorithm 12 Function to ensure boundary integrity

function CHECK_BOUNDARY_INTEGRITY_FUNCTION(X^{new})
    if $X^{new}[row, col] > UP[col]$ **then**
        $X^{new}[row, col] \leftarrow UP[col]$;
    end if
    if $X^{new}[row, col] < LB[col]$ **then**
        $X^{new}[row, col] \leftarrow LB[col]$;
    end if
end function

The selection of the best candidate solution by a greedy selection algorithm is parallelized in Algorithm 13.

Algorithm 13 Kernel for greedy selection

function GREEDY_SELECTION_KERNEL(X, X^{new})
    /* *Device code* */
    Determine $index$ using x dimension of $blockDim$, $blockIdx$, and $threadIdx$;
    if $index \leq popSize$ **then**
        if $f(X^{new}[index])$ is better than $f(X[index])$ **then**
            $X[index] \leftarrow X^{new}[index]$;
            $f(X[index]) \leftarrow f(X^{new}[index])$;
        end if
    end if
end function

Considering that the data structures supporting fitness values for both the newly generated candidate solutions and the current candidate solutions consist of single-dimensional arrays, each with a length of $popSize$, the kernel implementation requires solely a one-dimensional grid arrangement to conduct parallel comparisons.

Essentially, for concurrent determination of the optimal candidate solution, the greedy selection kernel operates using a total number of threads equivalent to $popSize$. The indexing of data and threads is managed by the variable $index$, which is determined using inherent variables with details about the hierarchical arrangement of blocks and threads.

The present GPU-based parallelization of Jaya distinguishes itself from the previous approaches by its efficient utilization of both the host and device during computation. Specifically, the host exclusively oversees the main loop, and data transfers between the host and device are minimized, never occurring during the most computationally demanding phases of the algorithm. Furthermore,

 doi:10.20944/preprints202405.0051.v1

the device directly accesses data without necessitating additional mapping or conversion. Notably, the approach to GPU grid arrangement and kernel invocations deviates from prior implementations. The dimensional structure of the grid and the determination of the necessary threads and blocks for computation are primarily driven by the size and structure of the data undergoing processing.

3.4. GPU-Based Acceleration of the Enhanced Jaya Algorithm

There is no mention in the existing literature of any previous GPU-based parallelization of the Enhanced Jaya (EJAYA) algorithm, which makes it impossible to directly compare the parallelization of the EJAYA algorithm proposed by the authors [29,30] and described in detail in this section with any other previous work.

While analyzing the differences between EJAYA and the original Jaya optimization algorithm, it becomes evident that EJAYA encompasses several additional steps and a more intricate formulation, primarily due to a more complex search method. Consequently, the GPU parallelization of EJAYA explained here demands a more elaborate approach than the original algorithm, requiring redesign and adaptation of its methods to meet the requisites and constraints of parallel processing. The overview of the GPU-based parallel EJAYA can be observed in Algorithm 14.

Algorithm 14 GPU-based parallel EJAYA

1: /* *Initialization* */
2: Initialize $numVar$, $popSize$ and $maxIter$; ▷ *Host*
3: $X \leftarrow$ GENERATE_INITIAL_POPULATION_KERNEL();
4: EVALUATE_FITNESS_VALUES_KERNEL(X);
5: $X^{old} \leftarrow X$
6: $i \leftarrow 1$; ▷ *Host*
7: /* *Main loop* */
8: **while** $i \leq maxIter$ **do** ▷ *Host*
9: Determine $X_{best,i}$ and $X_{worst,i}$;
10: DETERMINE_ATTRACT_POINTS_KERNEL();
11: **if** $P_{switch} \leq 0.5$ **then** ▷ *Host*
12: $X_i^{old} \leftarrow X_i$
13: **end if**
14: PERMUTE_POPULATION_KERNEL(X_i^{old});
15: $P_{select} \leftarrow$ DETERMINE_EXPLORATION_STRATEGY_KERNEL();
16: $X_i^{new} \leftarrow$ EJAYA-DRIVEN_POPULATION_UPDATE_KERNEL(X_i);
17: EVALUATE_FITNESS_VALUES_KERNEL(X_i^{new});
18: GREEDY_SELECTION_KERNEL(X_i, X_i^{new});
19: $i \leftarrow i + 1$; ▷ *Host*
20: **end while**
21: Output the best solution found and terminate; ▷ *Host and device*

According to the GPU-based parallelization strategy employed, all updated candidate solutions are generated simultaneously in parallel during each algorithm iteration. This means that some parts of the implementation need to be adjusted to ensure that all essential data for the LES and GES phases (Eqs. 5 and 8) are readily available for computation in the population update kernel.

Consequently, to perform the LES phase, the GPU-based EJAYA requires the prior determination of the upper and lower local attract points (PU and PL), as indicated on Eqs. 2 and 4. This procedure is described in Algorithm 15.

 doi:10.20944/preprints202405.0051.v1

Algorithm 15 Kernel for determining upper and lower attract points

```
1: function DETERMINE_ATTRACT_POINTS_KERNEL()
2:     /* Device code */
3:     Determine index using x dimension of blockDim, blockIdx, and threadIdx;
4:     if index ≤ numVar then
5:         PU[index] ← rand_PU × X_best[index] + (1 − rand_PU) × popMean[index];      ▷ Eq. (2)
6:         PL[index] ← rand_PL × X_worst[index] + (1 − rand_PL) × popMean[index];     ▷ Eq. (4)
7:     end if
8: end function
```

Both PU and PL are single-dimensional arrays with a length of the problem dimension ($numVar$). This workload is parallelized simultaneously using a one-dimensional grid arrangement with a number of threads equal to $numVar$, where each thread, denoted by the variable $index$, handles the computation of a specific element within the arrays. The formulation of PU and PL necessitates the determination of two random numbers ($rand_{PU}$ and $rand_{PL}$) and the population mean ($popMean$) prior to the kernel execution.

For the GES phase of the GPU-based EJAYA, the historical population (X^{old}) is determined using Eq. 6 and subsequently processed by Eq. 7. The determination of the switching probability P_{switch} is conducted by the host (specifically, in line 11 of Algorithm 14). This approach is preferable as it is more computationally efficient to handle a simple instruction such as an if statement in the host rather than by the device, considering the overhead related to kernel invocation. For the permutation of the historical population, the first step involves predetermining the random permutation of the position index of the candidate solutions within the population ($rand\_perm \leftarrow randomize([1, popSize])$). This permutation is then used as a template to shuffle the order of the candidates, as described in Algorithm 16.

Algorithm 16 Kernel for permuting the population

```
1: function PERMUTE_POPULATION_KERNEL(X^old)
2:     /* Device code */
3:     Determine index using x dimension of blockDim, blockIdx, and threadIdx;
4:     if index ≤ popSize then
5:         X^old[index] ← X^old[rand_perm[index]];
6:     end if
7: end function
```

Before updating the population, the exploration strategy, denoted as P_{select} in Eq. 9, needs to be determined. In the GPU-based EJAYA, where the newly generated candidate solutions X^{new} represent the candidates for the entire population in a single iteration, the P_{select} factor must correspond to the exploration strategy for the whole population. This ensures that the exploration strategy aligns with the parallel processing of the entire population. The procedure is illustrated in Algorithm 17.

Algorithm 17 Kernel for determining the exploration strategy

```
1: function DETERMINE_EXPLORATION_STRATEGY_KERNEL( )
2:     /* Device code */
3:     Determine index using x dimension of blockDim, blockIdx, and threadIdx;
4:     if index ≤ popSize then
5:         if rand() > 0.5 then
6:             P_select[index] ←'LES';
7:         else
8:             P_select[index] ←'GES';
9:         end if
10:    end if
11: end function
```

 doi:10.20944/preprints202405.0051.v1

The functionality required for performing both the LES and GES search strategies during the population update phase is facilitated through a unified kernel, as illustrated in Algorithm 18. In this kernel, the method for updating the population in the GES phase requires predefining an array of random numbers following a normal distribution called *randn*, with a length equal to *popSize*, in each iteration. This approach ensures a correct parallelization of Eq. 8, as the same random number is applied to update every dimension of a population candidate.

Algorithm 18 Kernel for population update (EJAYA)

function EJAYA-DRIVEN_POPULATION_UPDATE_KERNEL(X)
  / Device code */*
  Determine *row* using x dimension of *blockDim*, *blockIdx*, and *threadIdx*;
  Determine *col* using y dimension of *blockDim*, *blockIdx*, and *threadIdx*;
  if $row \leq popSize$ **and** $col \leq numVar$ **then**
    if $P_{select}[row] =$ *'LES'* **then** ▷ *LES*
      $X^{new}[row, col] \leftarrow X[row, col] + rand() \times (PU[col] - X[row, col]) - rand() \times (PL[col] - X[row, col])$; ▷ *Eq. (5)*
    else ▷ *GES*
      $X^{new}[row, col] \leftarrow X[row, col] + randn[row] \times (X^{old}[row, col] - X[row, col])$; ▷ *Eq. (8)*
    end if
    CHECK_BOUNDARY_INTEGRITY_FUNCTION(X^{new});
  end if
end function

3.5. GPU-Based Acceleration of the Rao Optimization Algorithms

Algorithm 19 presents a comprehensive characterization of a collective GPU-based parallelization of the three Rao algorithms recently proposed by the authors [31]. This implementation integrates the elementary methods constituting Rao-1, Rao-2, and Rao-3 heuristics into a singular and unified parallelization.

Algorithm 19 GPU-based parallel Rao-1, Rao-2, and Rao-3

/ Initialization */*
Initialize *numVar*, *popSize* and *maxIter*; ▷ *Host*
$X \leftarrow$ GENERATE_INITIAL_POPULATION_KERNEL();
EVALUATE_FITNESS_VALUES_KERNEL(X);
$i \leftarrow 1$; ▷ *Host*
/ Main loop */*
while $i \leq maxIter$ **do** ▷ *Host*
  Determine $X_{best,i}$ and $X_{worst,i}$;
  if *algorithm* = *'Rao-2'* **or** *'Rao-3'* **then**
    $t_i \leftarrow$ SELECT_RANDOM_SOLUTIONS_KERNEL();
  end if
  $X_i^{new} \leftarrow$ RAO-DRIVEN_POPULATION_UPDATE_KERNEL(X_i);
  EVALUATE_FITNESS_VALUES_KERNEL(X_i^{new});
  GREEDY_SELECTION_KERNEL(X_i, X_i^{new});
  $i \leftarrow i + 1$; ▷ *Host*
end while
Output the best solution found and terminate; ▷ *Host and device*

The three Rao algorithms employ a shared optimization approach, differing primarily in their population update strategies. Their fundamental algorithmic phases bear resemblance to those in Jaya. Notably, the main distinction arises in Rao-2 and Rao-3, where these algorithms necessitate selecting a random candidate solution from the population to compute the newly generated candidate solutions. Given the current parallelization strategy, this random candidate solution must be established in

 doi:10.20944/preprints202405.0051.v1

advance for all population candidates in each iteration before conducting the population update. This procedure is outlined in Algorithm 20.

Algorithm 20 Kernel for selecting random solutions from the population

```
function SELECT_RANDOM_SOLUTIONS_KERNEL()
    /* Device code */
    Determine index using x dimension of blockDim, blockIdx, and threadIdx;
    if index ≤ popSize then
        t[index] ← rand(1, popSize)
        while t[index] ≠ index do
            t[index] ← rand(1, popSize)
        end while
    end if
end function
```

The kernel is organized to operate using a one-dimensional grid, as its purpose is to generate a 1D array with a length equal to $popSize$ containing the indices of the random candidate solutions.

Algorithm 21 provides a consolidated method that generates the updated candidate solutions for all the Rao algorithms in a unified manner.

Algorithm 21 Kernel for population update (Rao algorithms)

```
function RAO-DRIVEN_POPULATION_UPDATE_KERNEL(X)
    /* Device code */
    Determine row using x dimension of blockDim, blockIdx, and threadIdx;
    Determine col using y dimension of blockDim, blockIdx, and threadIdx;
    if row ≤ popSize and col ≤ numVar then
        if algorithm = 'Rao-1' then                                          ▷ Eq. (10)
            X^new[row, col] ← X[row, col] + rand() × (X_best[col] − X_worst[col]);
        else if algorithm = 'Rao-2' then                                     ▷ Eq. (11)
            if f(X)[row] is better than f(X)[t[row]] then
                X^new[row, col] ← X[row, col] + rand() × (X_best[col] −
                X_worst[col]) + rand() × (abs(X[row, col]) − abs(X[t[row], col]));
            else
                X^new[row, col] ← X[row, col] + rand() × (X_best[col] −
                X_worst[col]) + rand() × (abs(X[t[row], col]) − abs(X[row, col]));
            end if
        else if algorithm = 'Rao-3' then                                     ▷ Eq. (12)
            if f(X)[row] is better than f(X)[t[row]] then
                X^new[row, col] ← X[row, col] + rand() × (X_best[col] −
                abs(X_worst[col])) + rand() × (abs(X[row, col]) − X[t[row], col]);
            else
                X^new[row, col] ← X[row, col] + rand() × (X_best[col] −
                abs(X_worst[col])) + rand() × (abs(X[t[row], col]) − X[row, col]);
            end if
        end if
        CHECK_BOUNDARY_INTEGRITY_FUNCTION(X^new);
    end if
end function
```

3.6. GPU-Based Acceleration of the BWP Algorithm

The main skeleton of the GPU-based parallel Best-Worst-Play (BWP) algorithm presented by the authors in [32] is shown in Algorithm 22.

 doi:10.20944/preprints202405.0051.v1

Algorithm 22 GPU-based parallel BWP algorithm.

1: */* Initialization */*
2: Initialize $numVar$, $popSize$ and $maxIter$; ▷ *Host*
3: $X \leftarrow$ GENERATE_INITIAL_POP_KERNEL();
4: EVALUATE_FITNESS_KERNEL(X);
5: $i \leftarrow 1$; ▷ *Host*
6: */* Main loop */*
7: **while** $i \leq maxIter$ **do** ▷ *Host*
8: */* Leverage the Jaya heuristic */*
9: Determine $X_{best,i}$ and $X_{worst,i}$;
10: $X_i^{new} \leftarrow$ JAYA-DRIVEN_POPULATION_UPDATE_KERNEL(X_i);
11: EVALUATE_FITNESS_KERNEL(X_i^{new});
12: GREEDY_SELECTION_KERNEL(X_i, X_i^{new});
13: */* Leverage the Rao-1 based heuristic */*
14: Determine $X_{best,i}$ and $X_{worst,i}$;
15: $X_i^{new} \leftarrow$ BWP-RAO1-BASED_POPULATION_UPDATE_KERNEL(X_i);
16: EVALUATE_FITNESS_KERNEL(X_i^{new});
17: GREEDY_SELECTION_KERNEL(X_i, X_i^{new});
18: $i \leftarrow i + 1$; ▷ *Host*
19: **end while**
20: Output the best solution found and terminate; ▷ *Host and device*

Although the BWP optimization strategy involves the consecutive use of two distinct algorithms (namely Jaya and a Rao-1 based heuristic), it is not possible to reduce the number of steps in the main loop. This limitation stems from the absence of overlapping methods, as each algorithmic phase is inherently sequential in nature.

The contents of Algorithm 23 detail the population update process within the Rao-1 based method.

Algorithm 23 Kernel for population update (BWP)

1: **function** BWP-RAO1-BASED_POPULATION_UPDATE_KERNEL(X)
2: */* Device code */*
3: Determine row using x dimension of $blockDim$, $blockIdx$, and $threadIdx$;
4: Determine col using y dimension of $blockDim$, $blockIdx$, and $threadIdx$;
5: **if** $row \leq popSize$ **and** $col \leq numVar$ **then**
6: $X^{new}[row, col] = X[row, col] + rand() \times (X_{best}[col] - |X_{worst}[col]|)$; ▷ *Eq. (13)*
7: CHECK_BOUNDARY_INTEGRITY_FUNCTION(X^{new});
8: **end if**
9: **end function**

3.7. A Novel GPU-Based Parallelization of the MaGI Algorithm

This section introduces a novel GPU-based parallelization of the Max–Min-Greedy-Interaction (MaGI) algorithm following the methodology outlined in this paper, as illustrated by Algorithm 24.

Since the MaGI algorithm employs two sequential metaphor-less heuristics in sequence, its approach to parallel implementation closely resembles that of BWP.

In accordance with the original algorithm (Algorithm 8), the first phase of the main loop of the GPU-based parallel MaGI algorithm involves employing the Jaya heuristic to generate an updated population, followed by further refinement using a Rao-2 based heuristic. This last heuristic requires the identification of random solutions from the population that have to be precomputed for all candidate solutions before population updates take place (line 15 of Algorithm 24), in accordance with the parallelization strategy presented for the original Rao-2 algorithm.

The GPU-based parallel implementation of the modified version of the Rao-2 heuristic is presented in Algorithm 25.

 doi:10.20944/preprints202405.0051.v1

Algorithm 24 GPU-based parallel MaGI algorithm

/* *Initialization* */
Initialize $numVar$, $popSize$ and $maxIter$; ▷ *Host*
$X \leftarrow$ GENERATE_INITIAL_POP_KERNEL();
EVALUATE_FITNESS_KERNEL(X);
$i \leftarrow 1$; ▷ *Host*
/* *Main loop* */
while $i \leq maxIter$ **do** ▷ *Host*
    /* *Leverage the Jaya heuristic* */
    Determine $X_{best,i}$ and $X_{worst,i}$;
    $X_i^{new} \leftarrow$ JAYA-DRIVEN_POPULATION_UPDATE_KERNEL(X_i);
    EVALUATE_FITNESS_KERNEL(X_i^{new});
    GREEDY_SELECTION_KERNEL(X_i, X_i^{new});
    Determine $X_{best,i}$ and $X_{worst,i}$;
    /* *Leverage the Rao-2 based heuristic* */
    $t_i \leftarrow$ SELECT_RANDOM_SOLUTIONS_KERNEL();
    $X_i^{new} \leftarrow$ MAGI-RAO2-BASED_POPULATION_UPDATE_KERNEL(X_i);
    EVALUATE_FITNESS_KERNEL(X_i^{new});
    GREEDY_SELECTION_KERNEL(X_i, X_i^{new});
    $i \leftarrow i + 1$; ▷ *Host*
end while
Output the best solution found and terminate; ▷ *Host and device*

Algorithm 25 Kernel for population update (MaGI)

function MAGI-RAO2-BASED_POPULATION_UPDATE_KERNEL(X)
    /* *Device code* */
    Determine row using x dimension of $blockDim$, $blockIdx$, and $threadIdx$;
    Determine col using y dimension of $blockDim$, $blockIdx$, and $threadIdx$;
    if $row \leq popSize$ **and** $col \leq numVar$ **then**
        if $f(X)[row]$ is better than $f(X)[t[row]]$ **then** ▷ *Eq. (14)*
            $X^{new}[row, col] \leftarrow X[row, col] + rand() \times (X_{best}[col] - abs(X_{worst}[col])) + rand() \times (X[row, col] - X[t[row], col])$;
        else
            $X^{new}[row, col] \leftarrow X[row, col] + rand() \times (X_{best}[col] - abs(X_{worst}[col])) + rand() \times (X[t[row], col] - X[row, col])$;
        end if
        CHECK_BOUNDARY_INTEGRITY_FUNCTION(X^{new});
    end if
end function

4. Computational Experiments

4.1. Experimental Setting and Implementation

Throughout this study, all computation times encompass the entire algorithmic process, from its initialization phase to reaching the stopping criterion and reporting the best solution found. This comprehensive approach ensures unbiased comparisons between the execution times of sequential and parallel implementations of the same algorithm. Specifically, in the case of parallel implementations running on the GPU, considerations such as algorithm initialization, data movements, overheads associated with kernel calls, and other sequential computation costs inherent in its heterogeneous approach are factored into their computational times.

The sequential and parallel implementations were written in Julia programming language [33] (version 1.9.4) using double precision floating-point arithmetic. Julia is an open-source and dynamic high-level language for scientific computation that shares syntax similarities with MATLAB, a widely adopted language in academia and industry for optimization problems. The parallel implementations

 doi:10.20944/preprints202405.0051.v1

leveraged the *CUDA.jl* package [34] (version 5.1.1) that serves as the primary gateway for programming NVIDIA GPUs in Julia.

To ensure a more rigorous and unbiased comparison of results, meticulous care was taken to implement the sequential and parallel algorithms in a highly standardized manner. This deliberate approach helps ensure a fair and accurate assessment of their performance.

The algorithmic parameters selected for this study encompassed a range of problem dimensions extending from 500 to 2000, with increments of 500. Additionally, the population size was determined to be 10 times the problem dimension, resulting in a range of population sizes from 5000 to 20 000, with increments of 5000. To improve the accuracy and reliability of experimental results, a total of 31 independent runs were carried out for every combination of algorithm, problem instance, and dimension before averaging the results, and the stopping criterion for each run was set to 1000 iterations.

Julia programming language utilizes a just-in-time (JIT) compilation process, translating high-level code into machine code [35]. This particular feature often results in longer execution times during the initial runtime of a Julia program. To ensure the integrity of the computational analysis, a warm-up phase was implemented that runs prior to testing. During this phase, a single run of each test combination is executed, and the results are discarded. Ultimately, this approach effectively prevents the inclusion of the delay caused by the JIT compiler during the first-time loading and compiling of Julia code.

Both the sequential and parallel implementations used the same parameters, ensuring a consistent evaluation across all experiments.

Sequential computation times of the tested algorithms serve as a reference point for performing a comparative analysis aimed at quantifying the degree of acceleration achieved by their corresponding GPU-based implementation.

4.2. Hardware Used in the Computational Experiments

The hardware setup for the sequential algorithm comprised the AMD Ryzen 9 5950X desktop CPU, featuring 16 cores, 32 threads, and 16 GB of DDR4 RAM. Despite the availability of the AMD Epyc 7643 server CPU, the AMD Ryzen 9 5950X was selected for the sequential tests due to its superior performance in our analysis. Although both CPUs are implementations of the same Zen 3 microarchitecture, the Ryzen 9 5950X, with its higher clock speeds (3.4 GHz base and 4.9 GHz boost compared to 2.3 GHz base and 3.6 GHz boost of the Epyc 7643 CPU), was able to achieve around 13.2% faster mean computational times. Higher clock speeds can have a significant impact on CPU performance [36], resulting in faster execution of instructions and improved throughput.

Performance analysis of GPU-based parallel algorithms was conducted on different GPUs with distinct hardware configurations. This approach was adopted to gain insights into the real-world applicability of the employed parallelization strategy, evaluating its performance and scalability across different architectures and available computing resources. The following GPU hardware was selected for the GPU-based computational experiments:

- NVIDIA GeForce RTX 3090 GPU (Ampere architecture) with 10 496 CUDA cores and 24 GB of GDDR6X VRAM;
- NVIDIA Tesla T4 GPU (Turing architecture) with 2560 CUDA cores and 16 GB of GDDR6 VRAM;
- NVIDIA Tesla V100S PCIe GPU (Volta architecture) with 5120 CUDA cores and 32 GB of HBM2 VRAM;
- NVIDIA A100 PCIe GPU (Ampere architecture) with 6912 CUDA cores and 80 GB of HBM2e VRAM.

Performance across the GPU devices will serve as benchmarks for evaluating algorithm efficiency and identifying potential bottlenecks. Initially, detailed performance analysis will primarily focus on the RTX 3090 and the A100 GPUs, both from the same Ampere architecture. While the former is a consumer-grade GPU, the latter is a professional-grade device. One of the main differences between

 doi:10.20944/preprints202405.0051.v1

these GPU classes is that the RTX 3090 was designed to have a high number of CUDA cores, whereas the A100 was designed for high memory bandwidth. This analysis aims to thoroughly evaluate the behavior of parallel algorithms in terms of performance and scalability across a spectrum of hardware configurations, assessing their adaptability to different available hardware resources.

4.3. Test Problems

The collection of SNLEs chosen to serve as benchmark problems represents the modeling of several real-world scenarios from a variety of scientific disciplines, including engineering, physics, and mathematics. These equations were specifically selected from existing literature due to their complex nature, encompassing a wide range of difficulty levels. Consequently, they present significant challenges for their resolution, in particular through traditional iterative numerical methods, also due to the computational demands involved.

Upon consideration of the analyzed algorithms, benchmark functions, and problem dimensions, a total of 280 distinct test setups were identified for 31 separate executions in both sequential and parallel implementations (with repetition for each tested GPU hardware).

The mathematical expressions for the selected benchmark functions are given below. All test problems demonstrate scalability in terms of problem dimension, emphasizing the growing demand for computer resources as the problem size increases.

Problem 1. *(Broyden tridiagonal function [37], $n = 500, 1000, 1500, 2000$)*

$$f_1(\boldsymbol{x}) = (3 - 2x_1)x_1 - 2x_2 + 1$$
$$f_n(\boldsymbol{x}) = (3 - 2x_n)x_n - x_{n-1} + 1$$
$$f_i(\boldsymbol{x}) = (3 - 2x_i)x_i - x_{i-1} - 2x_{i+1} + 1, \quad i = 2, \ldots, n-1$$
$$D = ([-1, 1], \ldots, [-1, 1])^T$$

Problem 2. *(Discrete boundary value function [37], $n = 500, 1000, 1500, 2000$)*

$$f_1(\boldsymbol{x}) = 2x_1 - x_2 + h^2(x_1 + h + 1)^3/2$$
$$f_n(\boldsymbol{x}) = 2x_n - x_{n-1} + h^2(x_n + nh + 1)^3/2$$
$$f_i(\boldsymbol{x}) = 2x_i - x_{i-1} - x_{i+1} + h^2(x_i + t_i + 1)^3/2, \quad i = 2, \ldots, n-1,$$
where $h = \frac{1}{n+1}$ *and* $t_i = ih$
$$D = ([0, 5], \ldots, [0, 5])^T$$

Problem 3. *(Extended Powell singular function [37], $n = 500, 1000, 1500, 2000$)*

$$f_{4i-3}(\boldsymbol{x}) = x_{4i-3} + 10x_{4i-2}$$
$$f_{4i-2}(\boldsymbol{x}) = \sqrt{5}(x_{4i-1} - x_{4i})$$
$$f_{4i-1}(\boldsymbol{x}) = (x_{4i-2} - 2x_{4i-1})^2$$
$$f_{4i}(\boldsymbol{x}) = \sqrt{10}(x_{4i-3} - x_{4i})^2, \quad i = 1, \ldots, 5$$
$$D = ([-100, 100], \ldots, [-100, 100])^T$$

Problem 4. *(Modified Rosenbrock function [38], $n = 500, 1000, 1500, 2000$)*

$$f_{2i-1}(\boldsymbol{x}) = \frac{1}{1 + \exp(-x_{2i-1})} - 0.73$$
$$f_{2i}(\boldsymbol{x}) = 10(x_{2i} - x_{2i-1}^2), \quad i = 1, \ldots, \tfrac{n}{2}$$
$$D = ([-10, 10], \ldots, [-10, 10])^T$$

Problem 5. *(Powell badly scaled function [38], $n = 500, 1000, 1500, 2000$)*

$$f_{2i-1}(\boldsymbol{x}) = 10^4 x_{2i-1}x_{2i} - 1$$
$$f_{2i}(\boldsymbol{x}) = \exp(-x_{2i-1}) + \exp(-x_{2i}) - 1.0001, \quad i = 1, \ldots, \tfrac{n}{2}$$
$$D = ([0, 100], \ldots, [0, 100])^T$$

Problem 6. *(Schubert–Broyden function [39], $n = 500, 1000, 1500, 2000$)*

 doi:10.20944/preprints202405.0051.v1

$f_1(\boldsymbol{x}) = (3 - x_1)x_1 + 1 - 2x_2$
$f_n(\boldsymbol{x}) = (3 - x_n)x_n + 1 - x_{n-1}$
$f_i(\boldsymbol{x}) = (3 - x_i)x_i + 1 - x_{i-1} - 2x_{i+1}, \quad i = 2, \ldots, n-1$
$D = ([-100, 100], \ldots, [-100, 100])^T$

Problem 7. *(Martínez function [40], $n = 500, 1000, 1500, 2000$)*
$f_1(\boldsymbol{x}) = (3 - 0.1x_1)x_1 + 1 - 2x_2 + x_1$
$f_n(\boldsymbol{x}) = (3 - 0.1x_n)x_n + 1 - 2x_{n-1} + x_n$
$f_i(\boldsymbol{x}) = (3 - 0.1x_i)x_i + 1 - x_{i-1} - 2x_{i+1} + x_1, \quad i = 2, \ldots, n-1$
$D = ([-100, 100], \ldots, [-100, 100])^T$

Problem 8. *(Extended Rosenbrock function [37], $n = 500, 1000, 1500, 2000$)*
$f_{2i-1}(\boldsymbol{x}) = 10(x_{2i} - x_{2i-1}^2)$
$f_{2i}(\boldsymbol{x}) = 1 - x_{2i-1}, \quad i = 1, \ldots, \frac{n}{2}$
$D = ([-100, 100], \ldots, [-100, 100])^T$

Problem 9. *(Bratu's problem [41], $n = 500, 1000, 1500, 2000$)*
$f_1(\boldsymbol{x}) = -2x_1 + x_2 + \alpha h^2 \exp(x_1)$
$f_n(\boldsymbol{x}) = x_{n-1} - 2x_n + \alpha h^2 \exp(x_n)$
$f_i(\boldsymbol{x}) = x_{i-1} - 2x_i + x_{i+1} + \alpha h^2 \exp(x_i), \quad i = 2, \ldots, n-1,$
where $h = \frac{1}{n+1}$ *and* $\alpha \geq 0$ *is a parameter; here* $\alpha = 3.5$
$D = ([-100, 100], \ldots, [-100, 100])^T$

Problem 10. *(The beam problem [41], $n = 500, 1000, 1500, 2000$)*
$f_1(\boldsymbol{x}) = -2x_1 + x_2 + \alpha h^2 \sin(x_1)$
$f_n(\boldsymbol{x}) = x_{n-1} - 2x_n + \alpha h^2 \sin(x_n)$
$f_i(\boldsymbol{x}) = x_{i-1} - 2x_i + x_{i+1} + \alpha h^2 \exp(x_i), \quad i = 2, \ldots, n-1,$
where $h = \frac{1}{n+1}$ *and* $\alpha \geq 0$ *is a parameter; here* $\alpha = 11$
$D = ([-100, 100], \ldots, [-100, 100])^T$

5. Results and Discussion

The subsequent discussion delves into the results stemming from the parallelization of the metaphor-less optimization algorithms discussed in this article in the GPU architecture. The primary emphasis lies in evaluating the attained speedup gains, understanding their behavior concerning increases in the problem dimension, and exploring different GPU hardware configurations. This aims to unravel the implications of leveraging parallel computational resources for optimizing algorithmic performance.

The speedup factor quantifies the performance improvements resulting from the parallel implementation and is defined as the ratio of the sequential to the parallel computation time. It is important to note that the speedup values presented in this analysis were calculated using the full computational times without rounding up.

Test data is systematically organized into tables, wherein test problems are categorized by dimension (i.e., population size and number of variables). The mean sequential computational times are presented in the column labeled *CPU*, while the results for parallel computation are organized into distinct columns based on the GPU hardware used for testing. All reported computational times pertain to the comprehensive execution of the algorithm, encompassing the entire duration from initiation to completion.

The primary emphasis in parallel computation is on data obtained with the RTX 3090 and A100 GPUs, which are presented in columns bearing the same names. The selection of the RTX 3090 GPU aligns with its consistent use in previous parallelization of the majority of algorithms discussed in this article, while the A100 GPU represents a highly relevant hardware option in the professional

 doi:10.20944/preprints202405.0051.v1

sector, and it is employed here to illustrate the scalability of the parallel implementations and its adeptness in harnessing additional GPU resources without necessitating adjustments to the underlying parallelization strategy.

Data obtained from the analysis of mean computational times for both sequential and parallel implementations, along with the corresponding speedup gains for each studied algorithm, were organized into tables and grouped based on similarities in algorithmic structure. Consequently, Table 1 presents computational results for the Jaya and EJAYA algorithms; the three Rao algorithms are showcased in Table 2, while the outcomes of the tests conducted with the BWP and MaGI algorithms are displayed in Table 3. Although Jaya and EJAYA are presented side by side, it is important to highlight that the EJAYA algorithm possesses a distinct structure compared to the other algorithms.

Table 1. Mean computational results for the Jaya and Enhanced Jaya algorithms. The time is in seconds, and the speedup gain for each GPU is shown in parentheses.

		Jaya			EJAYA		
Pop. (Vars)	**Prob. no.**	**CPU time**	**RTX 3090 time (gain)**	**A100 time (gain)**	**CPU time**	**RTX 3090 time (gain)**	**A100 time (gain)**
5000 (**500**)	1	39.90	0.59 (**67.6**)	0.33 (**121.9**)	54.98	1.56 (**35.3**)	1.54 (**35.8**)
	2	49.72	0.66 (**74.8**)	0.33 (**151.0**)	56.23	1.65 (**34.1**)	1.54 (**36.5**)
	3	44.69	0.53 (**83.6**)	0.28 (**160.7**)	56.78	1.54 (**36.9**)	1.50 (**37.7**)
	4	50.11	0.63 (**79.2**)	0.28 (**176.4**)	58.86	1.64 (**35.9**)	1.51 (**39.0**)
	5	49.33	0.70 (**70.2**)	0.31 (**158.9**)	63.58	1.72 (**36.9**)	1.51 (**42.2**)
	6	42.70	0.54 (**79.6**)	0.31 (**135.7**)	57.49	1.56 (**36.8**)	1.54 (**37.4**)
	7	40.87	0.55 (**74.7**)	0.31 (**129.9**)	52.99	1.56 (**33.9**)	1.54 (**34.4**)
	8	42.18	0.54 (**78.2**)	0.29 (**146.7**)	53.54	1.54 (**34.7**)	1.51 (**35.4**)
	9	55.34	0.73 (**75.3**)	0.32 (**175.4**)	63.19	1.78 (**35.5**)	1.53 (**41.2**)
	10	57.69	0.74 (**77.8**)	0.32 (**182.8**)	62.75	1.76 (**35.6**)	1.54 (**40.7**)
10 000 (**1000**)	1	152.28	1.07 (**141.8**)	0.75 (**202.5**)	195.02	3.31 (**58.9**)	3.38 (**57.6**)
	2	163.46	1.45 (**112.5**)	0.75 (**216.9**)	198.86	3.70 (**53.8**)	3.39 (**58.7**)
	3	150.11	0.98 (**153.7**)	0.67 (**225.1**)	212.96	3.25 (**65.5**)	3.30 (**64.6**)
	4	154.95	1.38 (**112.4**)	0.69 (**224.7**)	217.28	3.65 (**59.6**)	3.32 (**65.5**)
	5	169.35	1.65 (**102.5**)	0.79 (**213.4**)	230.65	3.93 (**58.7**)	3.33 (**69.3**)
	6	146.03	1.03 (**142.1**)	0.75 (**194.0**)	214.38	3.31 (**64.8**)	3.39 (**63.3**)
	7	129.48	1.07 (**121.4**)	0.75 (**172.1**)	200.85	3.34 (**60.2**)	3.39 (**59.3**)
	8	129.78	1.00 (**129.7**)	0.69 (**188.3**)	194.61	3.26 (**59.6**)	3.31 (**58.7**)
	9	196.08	1.83 (**107.2**)	0.75 (**260.4**)	241.98	4.15 (**58.3**)	3.38 (**71.6**)
	10	203.50	1.82 (**111.8**)	0.75 (**270.8**)	240.76	4.11 (**58.5**)	3.38 (**71.2**)
15 000 (**1500**)	1	370.98	1.92 (**193.0**)	1.28 (**289.7**)	459.26	5.65 (**81.2**)	5.38 (**85.4**)
	2	346.46	2.66 (**130.4**)	1.28 (**270.6**)	470.60	6.48 (**72.6**)	5.40 (**87.1**)
	3	324.86	1.65 (**197.1**)	1.12 (**290.9**)	497.39	5.51 (**90.3**)	5.24 (**94.9**)
	4	339.46	2.57 (**132.1**)	1.15 (**294.4**)	525.25	6.49 (**81.0**)	5.28 (**99.5**)
	5	381.45	3.17 (**120.3**)	1.33 (**285.9**)	549.86	7.00 (**78.5**)	5.29 (**103.9**)
	6	319.96	1.77 (**180.4**)	1.28 (**249.6**)	497.15	5.62 (**88.5**)	5.40 (**92.0**)
	7	287.13	1.85 (**155.4**)	1.28 (**223.8**)	474.74	5.71 (**83.1**)	5.41 (**87.7**)
	8	292.39	1.70 (**171.7**)	1.15 (**255.3**)	462.30	5.57 (**83.1**)	5.27 (**87.7**)
	9	439.23	3.56 (**123.4**)	1.28 (**342.4**)	565.03	7.50 (**75.4**)	5.41 (**104.5**)
	10	457.05	3.54 (**129.0**)	1.28 (**356.7**)	567.82	7.41 (**76.7**)	5.41 (**105.0**)

 doi:10.20944/preprints202405.0051.v1

Table 1. *Cont.*

Pop. (Vars)	Prob. no.	Jaya			EJAYA		
		CPU time	RTX 3090 time (gain)	A100 time (gain)	CPU time	RTX 3090 time (gain)	A100 time (gain)
20 000 (**2000**)	1	737.74	2.82 (**261.5**)	1.93 (**383.0**)	949.89	8.74 (**108.7**)	7.59 (**125.1**)
	2	689.07	4.21 (**163.6**)	1.93 (**357.5**)	950.15	10.19 (**93.2**)	7.63 (**124.5**)
	3	729.06	2.46 (**295.8**)	1.65 (**443.0**)	977.18	8.48 (**115.3**)	7.36 (**132.7**)
	4	744.16	4.08 (**182.5**)	1.71 (**434.7**)	1030.45	10.21 (**100.9**)	7.42 (**138.9**)
	5	878.30	5.13 (**171.1**)	1.97 (**444.8**)	1119.75	11.01 (**101.7**)	7.46 (**150.1**)
	6	714.79	2.68 (**267.2**)	1.93 (**371.0**)	975.85	8.76 (**111.5**)	7.63 (**127.8**)
	7	674.34	2.82 (**239.4**)	1.93 (**350.0**)	919.19	8.83 (**104.1**)	7.64 (**120.4**)
	8	678.22	2.57 (**264.3**)	1.69 (**400.4**)	932.70	8.55 (**109.1**)	7.40 (**126.0**)
	9	974.28	5.83 (**167.0**)	1.93 (**505.3**)	1106.00	11.78 (**93.9**)	7.64 (**144.8**)
	10	904.68	5.81 (**155.8**)	1.93 (**468.5**)	1105.26	11.98 (**92.2**)	7.64 (**144.7**)

Findings reveal the superior efficiency of GPU-based algorithms compared to their corresponding sequential implementations. In all algorithms, the speedup gains increased proportionally with the growth of the problem dimension. Consequently, the lower speedup gains were observed in the smallest dimensions, while the most significant speedups were achieved in the largest dimensions. This expectation is rooted in the parallelization strategy employed for GPU-based algorithms, wherein the allocation of GPU processing capabilities (expressed in terms of the number of CUDA cores, the block size, and the number of blocks) is adjusted automatically according to the problem dimensionality and inherent characteristics of the GPU.

Regarding the GPU hardware, the A100 GPU demonstrated speedup gains ranging from a minimum of 34.4× in the EJAYA algorithm to a maximum of 561.8× in the Rao-3, achieving a global mean speedup (across all algorithms, problems, and dimensions) of 231.2×. This performance surpassed that of the RTX 3090 GPU, which attained speedups ranging from 33.9× in the EJAYA algorithm to 295.8× with the Jaya algorithm, resulting in a global mean speedup of 127.9×, approximately 44.7% lower than that of the A100 GPU.

Despite having fewer 3584 CUDA cores when compared to the RTX 3090 GPU (a reduction of 34.1%), the A100 GPU manages to perform more computational work by leveraging its significantly wider memory bus of 5120 bits, in contrast to the 384 bits in the RTX 3090, while sharing the same architecture. This implies that each CUDA core in the A100 GPU is able to operate more efficiently than its counterpart in the RTX 3090, mainly due to its enhanced memory subsystem, particularly under this type of workload. This outcome aligns with expectations, considering that factors such as the parallelization strategy, GPU architecture, the number of available CUDA cores, and the width of the GPU memory bus are pivotal for achieving optimal performance in CUDA programming.

The GPU-based algorithm that most effectively leveraged the parallel processing capabilities of the GPU hardware was the Rao-3 algorithm, with an average speedup across all problems and dimensions of 169.9× for the RTX 3090 GPU and 290.0× for the A100. In contrast, the GPU-based EJAYA algorithm demonstrated the more modest, although still relevant, speedup values, with an average speedup of 69.9× for the first GPU and 82.6× with the second. This observed result could be attributed to the parallelization strategy employed for the EJAYA algorithm, which faced challenges in attaining a similar level of efficiency as the remaining algorithms. Additionally, inherent characteristics of the EJAYA algorithm, such as the utilization of a more complex exploration method involving different strategies requiring switching and selecting probabilities, mean historical solutions, and permutation of population elements, contribute to its distinct performance dynamics.

 doi:10.20944/preprints202405.0051.v1

Table 2. Mean computational results for the Rao algorithms. The time is in seconds, and the speedup gain for each GPU is shown in parentheses.

		Rao-1			Rao-2			Rao-3		
Pop. (Vars)	**Prob. no.**	**CPU time**	**RTX 3090 time (gain)**	**A100 time (gain)**	**CPU time**	**RTX 3090 time (gain)**	**A100 time (gain)**	**CPU time**	**RTX 3090 time (gain)**	**A100 time (gain)**
5000 (**500**)	1	24.03	0.55 (**43.3**)	0.32 (**74.2**)	55.51	0.67 (**82.6**)	0.43 (**129.6**)	62.39	0.67 (**93.4**)	0.42 (**147.6**)
	2	27.01	0.66 (**41.1**)	0.33 (**82.0**)	50.78	0.76 (**67.0**)	0.42 (**120.5**)	76.75	0.75 (**101.8**)	0.42 (**183.2**)
	3	26.37	0.54 (**49.2**)	0.28 (**94.8**)	61.59	0.66 (**93.9**)	0.40 (**154.5**)	64.46	0.64 (**100.0**)	0.40 (**163.1**)
	4	30.90	0.63 (**48.8**)	0.28 (**109.1**)	56.52	0.75 (**75.7**)	0.40 (**143.0**)	79.82	0.75 (**107.0**)	0.39 (**206.2**)
	5	34.11	0.70 (**48.8**)	0.31 (**108.6**)	60.58	0.80 (**76.0**)	0.42 (**143.6**)	70.70	0.80 (**87.9**)	0.42 (**167.7**)
	6	25.76	0.55 (**46.5**)	0.32 (**81.7**)	58.38	0.66 (**88.2**)	0.45 (**129.4**)	59.20	0.66 (**89.1**)	0.45 (**132.7**)
	7	28.29	0.55 (**51.3**)	0.32 (**89.5**)	60.83	0.68 (**90.0**)	0.44 (**138.4**)	63.15	0.67 (**93.9**)	0.44 (**144.3**)
	8	29.80	0.55 (**54.5**)	0.29 (**103.8**)	61.48	0.65 (**94.3**)	0.41 (**151.4**)	65.34	0.66 (**99.6**)	0.40 (**162.8**)
	9	35.18	0.73 (**47.9**)	0.31 (**112.2**)	68.65	0.84 (**81.8**)	0.38 (**182.2**)	74.35	0.84 (**88.1**)	0.37 (**200.0**)
	10	32.55	0.74 (**43.9**)	0.32 (**103.1**)	65.59	0.84 (**78.4**)	0.38 (**173.7**)	73.43	0.83 (**88.2**)	0.37 (**197.0**)
10 000 (**1000**)	1	82.91	1.05 (**78.6**)	0.75 (**110.4**)	219.10	1.39 (**157.5**)	1.11 (**198.1**)	214.49	1.39 (**153.8**)	1.10 (**195.0**)
	2	90.39	1.41 (**64.2**)	0.75 (**119.9**)	204.32	1.75 (**116.8**)	1.11 (**183.9**)	283.81	1.76 (**161.1**)	1.11 (**256.4**)
	3	88.50	0.97 (**90.8**)	0.67 (**132.6**)	234.66	1.32 (**177.7**)	1.03 (**227.3**)	248.38	1.32 (**188.7**)	1.03 (**241.1**)
	4	101.40	1.38 (**73.5**)	0.69 (**146.9**)	227.08	1.72 (**132.1**)	1.04 (**218.1**)	292.75	1.72 (**170.3**)	1.04 (**281.8**)
	5	109.02	1.67 (**65.3**)	0.79 (**137.4**)	237.69	1.98 (**120.2**)	1.07 (**221.3**)	276.37	1.97 (**140.1**)	1.07 (**258.4**)
	6	86.80	1.04 (**83.8**)	0.75 (**115.9**)	227.92	1.36 (**167.7**)	1.13 (**202.5**)	231.80	1.37 (**169.1**)	1.12 (**206.7**)
	7	95.58	1.06 (**90.0**)	0.75 (**127.9**)	233.46	1.41 (**165.1**)	1.12 (**208.3**)	239.59	1.42 (**169.2**)	1.12 (**214.0**)
	8	101.57	1.00 (**102.0**)	0.69 (**147.4**)	236.84	1.34 (**176.8**)	1.05 (**225.9**)	248.91	1.34 (**185.8**)	1.05 (**237.9**)
	9	130.21	1.83 (**71.2**)	0.75 (**172.8**)	263.01	2.07 (**126.8**)	0.97 (**272.4**)	292.20	2.08 (**140.6**)	0.97 (**300.2**)
	10	112.72	1.84 (**61.3**)	0.76 (**149.2**)	242.10	2.11 (**115.0**)	0.97 (**249.8**)	290.69	2.11 (**137.6**)	0.96 (**301.3**)

 doi:10.20944/preprints202405.0051.v1

Table 2. *Cont.*

Pop. (Vars)	Prob. no.	Rao-1			Rao-2			Rao-3		
		CPU time	RTX 3090 time (gain)	A100 time (gain)	CPU time	RTX 3090 time (gain)	A100 time (gain)	CPU time	RTX 3090 time (gain)	A100 time (gain)
15 000 (**1500**)	1	190.52	1.82 (**104.7**)	1.28 (**148.8**)	515.54	2.56 (**201.1**)	1.94 (**265.5**)	507.35	2.57 (**197.4**)	1.94 (**262.0**)
	2	210.81	2.65 (**79.6**)	1.28 (**164.6**)	472.17	3.34 (**141.2**)	1.94 (**243.6**)	620.31	3.34 (**185.5**)	1.94 (**320.4**)
	3	188.79	1.65 (**114.7**)	1.12 (**169.1**)	536.08	2.38 (**225.0**)	1.78 (**300.3**)	578.01	2.38 (**242.6**)	1.78 (**324.5**)
	4	221.14	2.58 (**85.8**)	1.15 (**191.7**)	520.62	3.25 (**160.2**)	1.79 (**290.6**)	651.36	3.26 (**199.9**)	1.79 (**364.6**)
	5	230.39	3.19 (**72.3**)	1.34 (**171.7**)	548.90	3.84 (**142.8**)	1.87 (**294.2**)	631.89	3.85 (**164.1**)	1.86 (**339.6**)
	6	189.74	1.77 (**107.2**)	1.28 (**147.8**)	525.59	2.49 (**211.1**)	1.95 (**269.4**)	539.88	2.49 (**216.6**)	1.95 (**277.2**)
	7	209.67	1.86 (**113.0**)	1.28 (**163.5**)	537.68	2.59 (**207.3**)	1.96 (**274.2**)	564.67	2.66 (**212.6**)	1.96 (**288.2**)
	8	218.59	1.69 (**129.2**)	1.15 (**190.9**)	543.32	2.42 (**224.6**)	1.80 (**301.1**)	558.39	2.42 (**231.1**)	1.80 (**310.4**)
	9	295.56	3.57 (**82.9**)	1.28 (**230.5**)	585.36	3.85 (**152.1**)	1.48 (**396.4**)	660.64	3.94 (**167.5**)	1.52 (**435.6**)
	10	252.91	3.56 (**71.0**)	1.28 (**197.4**)	575.19	3.86 (**149.0**)	1.47 (**392.4**)	663.25	3.85 (**172.4**)	1.44 (**461.2**)
20 000 (**2000**)	1	362.13	2.81 (**128.8**)	1.93 (**188.0**)	961.60	4.08 (**235.5**)	2.99 (**321.4**)	948.69	4.07 (**232.8**)	2.99 (**317.6**)
	2	426.73	4.26 (**100.2**)	1.93 (**221.4**)	882.08	5.42 (**162.7**)	2.99 (**295.3**)	1114.94	5.43 (**205.2**)	2.98 (**374.1**)
	3	351.16	2.46 (**142.5**)	1.65 (**213.4**)	1001.60	3.75 (**267.3**)	2.72 (**367.8**)	1043.39	3.75 (**278.2**)	2.72 (**383.4**)
	4	458.26	4.08 (**112.2**)	1.71 (**267.7**)	952.20	5.27 (**180.6**)	2.76 (**345.4**)	1113.26	5.27 (**211.1**)	2.75 (**404.2**)
	5	481.20	5.13 (**93.7**)	1.97 (**243.7**)	1067.28	6.32 (**168.9**)	2.88 (**370.2**)	1185.69	6.32 (**187.7**)	2.88 (**411.8**)
	6	363.18	2.67 (**136.1**)	1.93 (**188.5**)	1029.87	3.95 (**260.8**)	2.99 (**344.2**)	1008.06	3.95 (**255.0**)	2.99 (**337.1**)
	7	364.72	2.81 (**129.9**)	1.93 (**189.3**)	993.69	4.13 (**240.6**)	3.01 (**330.3**)	965.57	4.18 (**230.9**)	3.01 (**321.1**)
	8	370.64	2.56 (**145.0**)	1.69 (**218.8**)	978.14	3.83 (**255.2**)	2.76 (**354.5**)	989.94	3.83 (**258.6**)	2.75 (**359.7**)
	9	564.40	5.83 (**96.9**)	1.93 (**292.7**)	1171.44	6.48 (**180.9**)	2.23 (**524.3**)	1206.58	6.43 (**187.5**)	2.21 (**546.7**)
	10	525.56	5.83 (**90.1**)	1.93 (**272.2**)	1078.68	6.45 (**167.2**)	2.21 (**489.2**)	1258.08	6.50 (**193.4**)	2.24 (**561.8**)

 doi:10.20944/preprints202405.0051.v1

Table 3. Mean computational results for the Jaya and Enhanced Jaya algorithms. The time is in seconds, and the speedup gain for each GPU is shown in parentheses.

		BWP			MaGI		
Pop. (Vars)	**Prob. no.**	**CPU time**	**RTX 3090 time (gain)**	**A100 time (gain)**	**CPU time**	**RTX 3090 time (gain)**	**A100 time (gain)**
5000 (**500**)	1	83.04	1.00 (**83.3**)	0.53 (**156.0**)	107.64	1.23 (**87.4**)	0.72 (**148.7**)
	2	79.48	1.34 (**59.4**)	0.65 (**121.5**)	117.63	1.41 (**83.2**)	0.76 (**155.2**)
	3	83.83	1.03 (**81.6**)	0.56 (**150.4**)	94.22	1.03 (**91.7**)	0.52 (**180.4**)
	4	88.05	1.30 (**68.0**)	0.58 (**151.0**)	116.55	1.33 (**87.7**)	0.63 (**185.6**)
	5	84.39	1.37 (**61.7**)	0.58 (**145.6**)	123.60	1.43 (**86.4**)	0.65 (**191.4**)
	6	74.44	1.01 (**73.9**)	0.54 (**137.4**)	93.03	1.18 (**78.6**)	0.69 (**135.6**)
	7	83.18	1.02 (**81.7**)	0.53 (**157.0**)	90.15	1.20 (**74.9**)	0.69 (**130.1**)
	8	64.09	1.04 (**61.4**)	0.54 (**118.2**)	91.30	1.17 (**78.2**)	0.65 (**141.0**)
	9	92.16	1.28 (**71.9**)	0.43 (**214.1**)	120.75	1.46 (**82.9**)	0.56 (**216.4**)
	10	98.07	1.28 (**76.5**)	0.44 (**224.9**)	128.80	1.46 (**88.0**)	0.58 (**221.1**)
10 000 (**1000**)	1	266.94	1.91 (**139.8**)	1.19 (**223.4**)	318.06	2.51 (**126.6**)	1.83 (**173.6**)
	2	265.09	2.98 (**89.0**)	1.59 (**166.8**)	452.80	3.23 (**140.0**)	1.87 (**241.7**)
	3	280.87	1.99 (**141.2**)	1.38 (**204.1**)	317.79	2.03 (**156.4**)	1.14 (**278.1**)
	4	300.54	2.87 (**104.8**)	1.44 (**208.6**)	419.10	3.05 (**137.4**)	1.51 (**276.6**)
	5	291.19	3.25 (**89.7**)	1.40 (**207.4**)	447.20	3.65 (**122.6**)	1.77 (**253.2**)
	6	248.07	1.88 (**132.3**)	1.22 (**204.2**)	334.23	2.33 (**143.3**)	1.62 (**206.8**)
	7	291.98	1.96 (**149.0**)	1.21 (**241.2**)	320.02	2.47 (**129.4**)	1.70 (**187.7**)
	8	225.04	1.85 (**121.8**)	1.10 (**204.4**)	329.33	2.34 (**140.7**)	1.58 (**207.9**)
	9	293.84	3.22 (**91.3**)	0.92 (**318.8**)	409.07	3.74 (**109.4**)	1.26 (**325.9**)
	10	340.01	3.22 (**105.5**)	0.89 (**381.0**)	454.26	3.47 (**130.8**)	1.27 (**357.2**)
15 000 (**1500**)	1	550.07	3.39 (**162.4**)	2.06 (**267.5**)	789.59	4.50 (**175.6**)	3.18 (**247.9**)
	2	581.02	5.48 (**106.0**)	2.65 (**218.9**)	1008.38	6.14 (**164.2**)	3.24 (**310.8**)
	3	614.33	3.40 (**180.8**)	2.30 (**267.1**)	665.81	3.53 (**188.5**)	2.10 (**317.6**)
	4	658.30	5.32 (**123.8**)	2.41 (**273.6**)	878.04	5.77 (**152.2**)	2.61 (**336.7**)
	5	637.82	6.20 (**102.8**)	2.34 (**272.8**)	993.52	7.12 (**139.5**)	3.15 (**315.6**)
	6	563.17	3.28 (**171.6**)	2.06 (**273.3**)	750.12	4.09 (**183.6**)	2.77 (**270.5**)
	7	598.77	3.47 (**172.3**)	2.06 (**291.4**)	740.67	4.41 (**167.8**)	3.00 (**247.1**)
	8	484.28	3.18 (**152.1**)	1.83 (**264.7**)	746.81	4.12 (**181.1**)	2.72 (**274.7**)
	9	596.64	6.53 (**91.4**)	1.69 (**352.9**)	869.23	6.94 (**125.2**)	2.17 (**401.2**)
	10	755.36	6.59 (**114.7**)	1.72 (**439.1**)	1002.26	7.27 (**137.8**)	2.23 (**450.0**)
20 000 (**2000**)	1	1190.73	5.37 (**221.9**)	3.20 (**372.5**)	1704.56	7.07 (**241.2**)	4.86 (**350.4**)
	2	1280.47	8.78 (**145.9**)	3.98 (**321.9**)	1956.22	9.95 (**196.6**)	4.94 (**395.9**)
	3	1299.94	5.21 (**249.4**)	3.40 (**382.4**)	1442.29	5.75 (**250.8**)	3.37 (**427.5**)
	4	1338.01	8.50 (**157.4**)	3.58 (**373.4**)	1732.39	9.31 (**186.1**)	4.00 (**433.2**)
	5	1391.69	10.30 (**135.1**)	3.57 (**389.5**)	1929.49	11.68 (**165.2**)	4.77 (**404.7**)
	6	1231.50	5.09 (**241.9**)	3.21 (**383.9**)	1503.84	6.43 (**233.9**)	4.33 (**347.1**)
	7	1147.42	5.43 (**211.4**)	3.21 (**357.6**)	1515.59	6.95 (**218.0**)	4.65 (**326.1**)
	8	1160.41	4.92 (**235.9**)	2.78 (**418.0**)	1501.78	6.46 (**232.4**)	4.11 (**365.0**)
	9	1293.98	11.03 (**117.3**)	3.01 (**429.4**)	1690.29	11.69 (**144.6**)	3.23 (**523.4**)
	10	1436.48	11.04 (**130.1**)	2.93 (**490.7**)	1798.28	11.75 (**153.1**)	3.28 (**548.7**)

Upon analyzing the results obtained for the Rao algorithms (refer to Table 2), a clear pattern becomes apparent. The Rao-1 algorithm, characterized by the simplest mathematical model among the three, exhibited the shortest computational time, resulting in an average of 194.23 seconds for the sequential algorithm (executed on the CPU). Conversely, Rao-2 and Rao-3 required approximately 147.9% more time on average than Rao-1 to complete identical tests, with a respective duration of 460.08 s and 502.96 s.

 doi:10.20944/preprints202405.0051.v1

In a comparative evaluation, the GPU-based implementations demonstrated superior scalability amidst the increasing complexity of the Rao algorithms when juxtaposed with their sequential counterparts. With the RTX 3090 GPU, the Rao-1 algorithm yielded a global average of 2.06 seconds, while Rao-2 and Rao-3 exhibited an increase of approximately 26.7% (2.60 s for Rao-2 and 2.61 s for Rao-3). Employing the A100 GPU, the difference in the global average between Rao-1 (1.04 s) and both Rao-2 and Rao-3 (1.51 s and 1.50 s) was approximately 45.1% higher. Although this denotes a superior growth rate in computational time when compared with the RTX 3090 GPU, it is noticeable that the A100 GPU achieved an average speedup that was 73.7% higher.

This enhanced scalability of the GPU-based algorithm is further reflected in the speedup values attained across the various Rao algorithms. The algorithm with lower computational demands achieved comparatively lower speedup gains. Specifically, for the GPU-based Rao-1 algorithm running on the RTX 3090, the average speedup was 84.8×, escalating to 155.4× for Rao-2 and 169.9× for Rao-3. Comparing to Rao-1, this represents an increase in speedup of 83.3% and 100.3%, respectively. A similar trend is observed with the A100 GPU, where average speedup gains for the Rao algorithms were 159.8×, 261.1×, and 290.0×. This denotes a comparative increase in speedup achieved with Rao-2 and Rao-3 over Rao-1 of 63.4% and 81.5%, respectively.

A similar scenario unfolds with the BWP and MaGI algorithms (Table 3), where the speedup attained by the GPU-based implementation increases when transitioning to a more computationally demanding algorithm. Despite both algorithms being Jaya-based, the MaGI algorithm features a slightly more intricate mathematical model, being based on Rao-2 as opposed to Rao-1 in the case of the BWP algorithm. Consequently, the sequential implementation of the MaGI algorithm resulted in a 34.1% increase in global average computation time compared to BWP.

When comparing both algorithms using the GPU-based implementation, the observed increases were smaller. Utilizing the RTX 3090, the increase was approximately 14.7%, and it rose to approximately 26.5% with the A100 GPU. Consistent with previous findings, while the A100 exhibits a greater growth in computational times compared to the RTX 3090 GPU when transitioning to a more demanding algorithm, the speedups achieved by the A100 were superior, resulting in an average increase of 104.5% when considering both the BWP and MaGI algorithms.

The behavior of both sequential and parallel implementations of the studied metaphor-less algorithms across the different problem dimensions, along with the achieved speedups, is depicted in Figures 3 through 9.

While profiling the different test problems and specifically examining the top three most and least computationally demanding in terms of average execution time, certain patterns become evident. Results from the sequential implementation of the algorithms reveal that problems 5, 9, and 10 consistently rank among the most demanding, with the exception of problem 9, which does not appear in the top three most demanding for the BWP and MaGi algorithms. In parallel testing, the RTX 3090 GPU exhibits a similar pattern, consistently ranking problems 5, 9, and 10 as the most computationally demanding across all algorithms. However, when using the A100 GPU, no similar or consistent pattern emerges. No single problem ranks as the most demanding of all algorithms, but problems 1, 2, and 5 typically rank among the slowest. Analyzing the test problems per individual algorithm, problem 5 appears as one of the slowest in both sequential and parallel implementations for most tested algorithms. The exception is in EJAYA, Rao-2, and Rao-3, where problem 5 does not rank among the top three slowest problems when computing with the A100 GPU.

 doi:10.20944/preprints202405.0051.v1

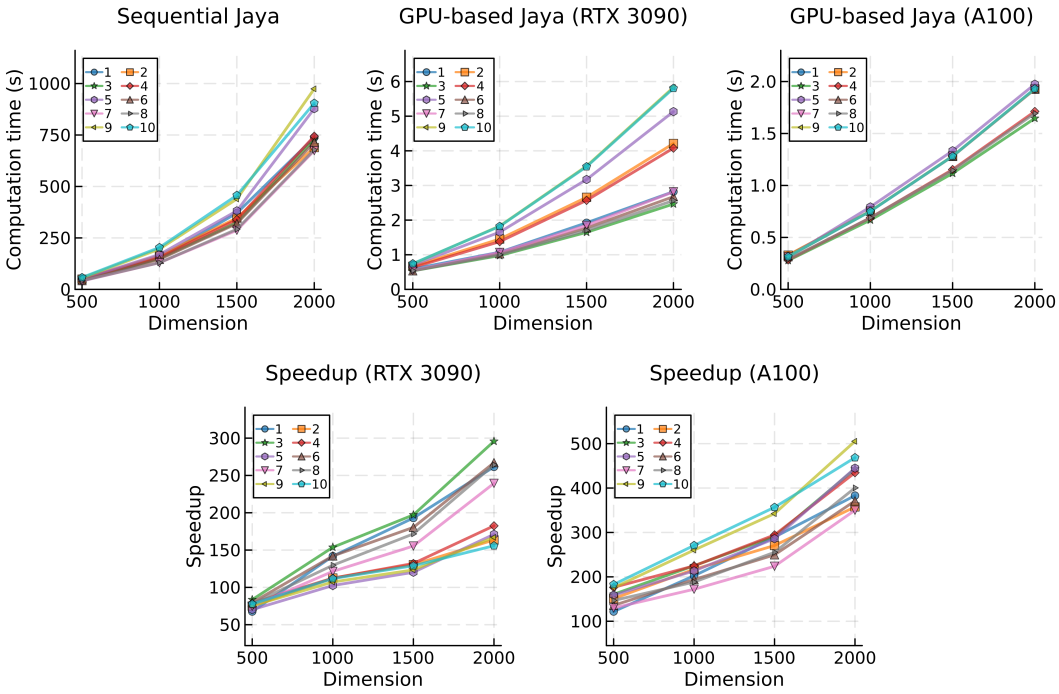

Figure 3. Mean computational times (top) and speedup (bottom) for the Jaya algorithm, categorized by problem and dimension.

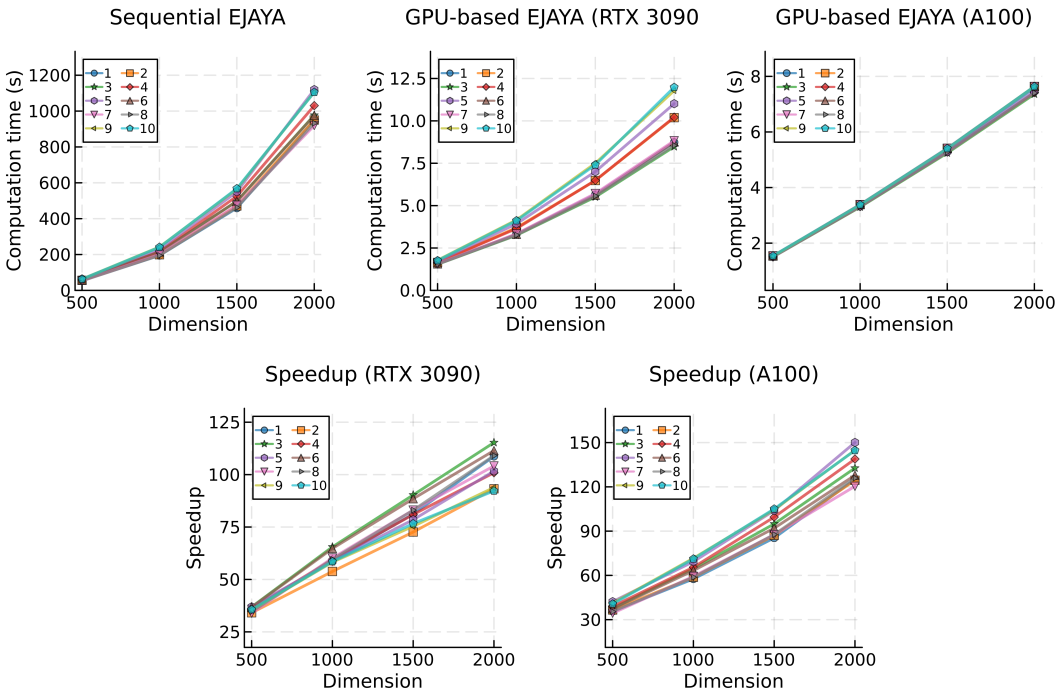

Figure 4. Mean computational times (top) and speedup (bottom) for the EJAYA algorithm, categorized by problem and dimension.

 doi:10.20944/preprints202405.0051.v1

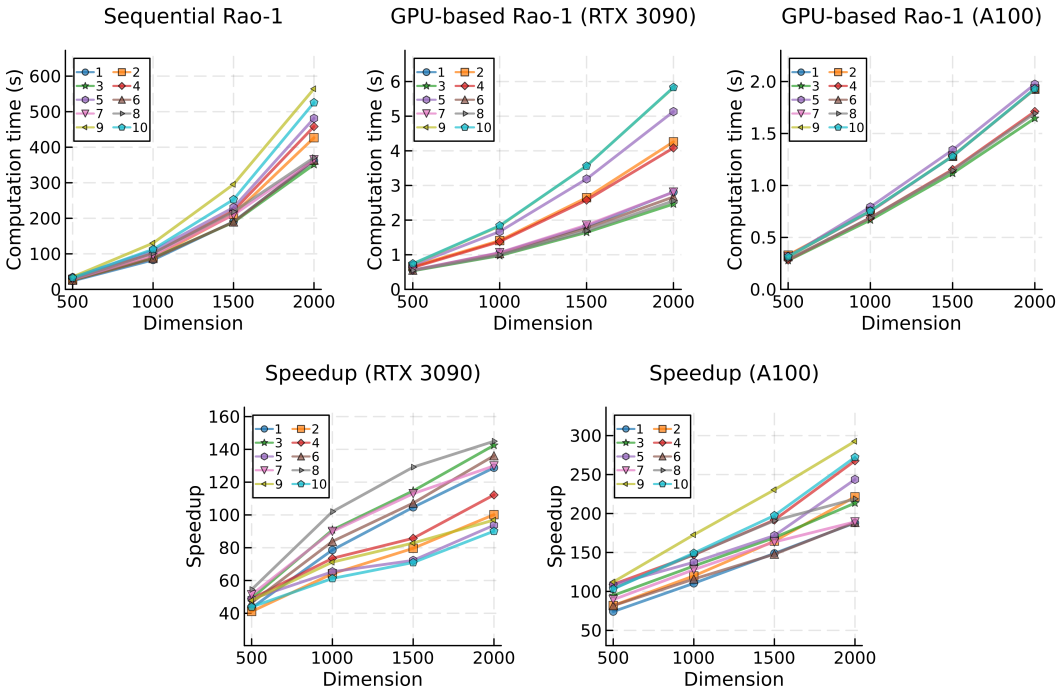

Figure 5. Mean computational times (top) and speedup (bottom) for the Rao-1 algorithm, categorized by problem and dimension.

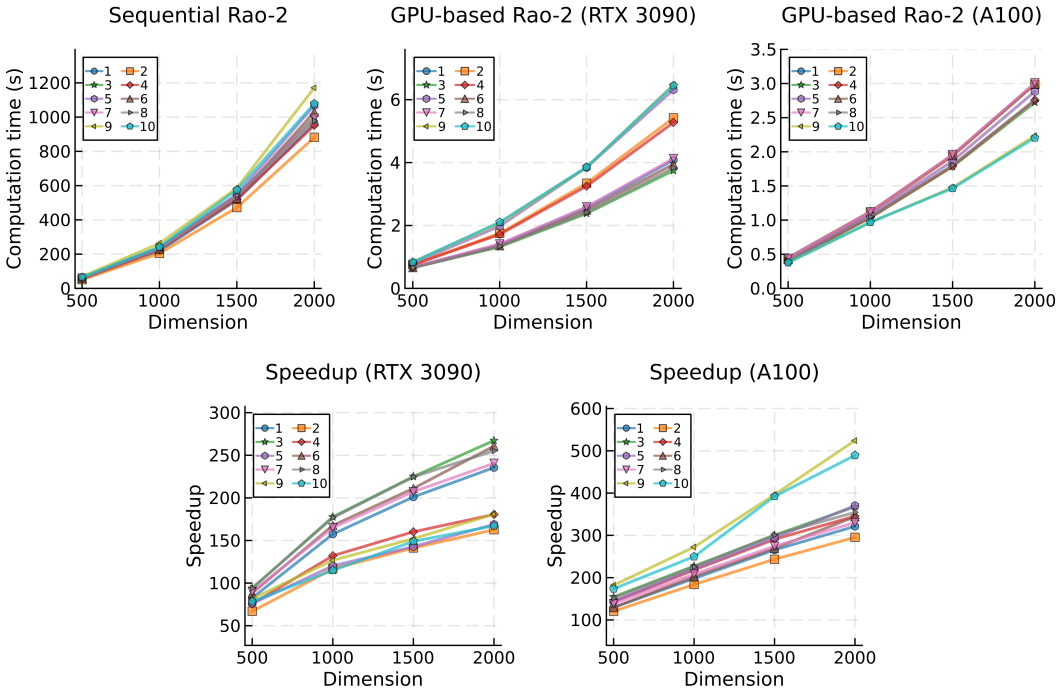

Figure 6. Mean computational times (top) and speedup (bottom) for the Rao-2 algorithm, categorized by problem and dimension.

 doi:10.20944/preprints202405.0051.v1

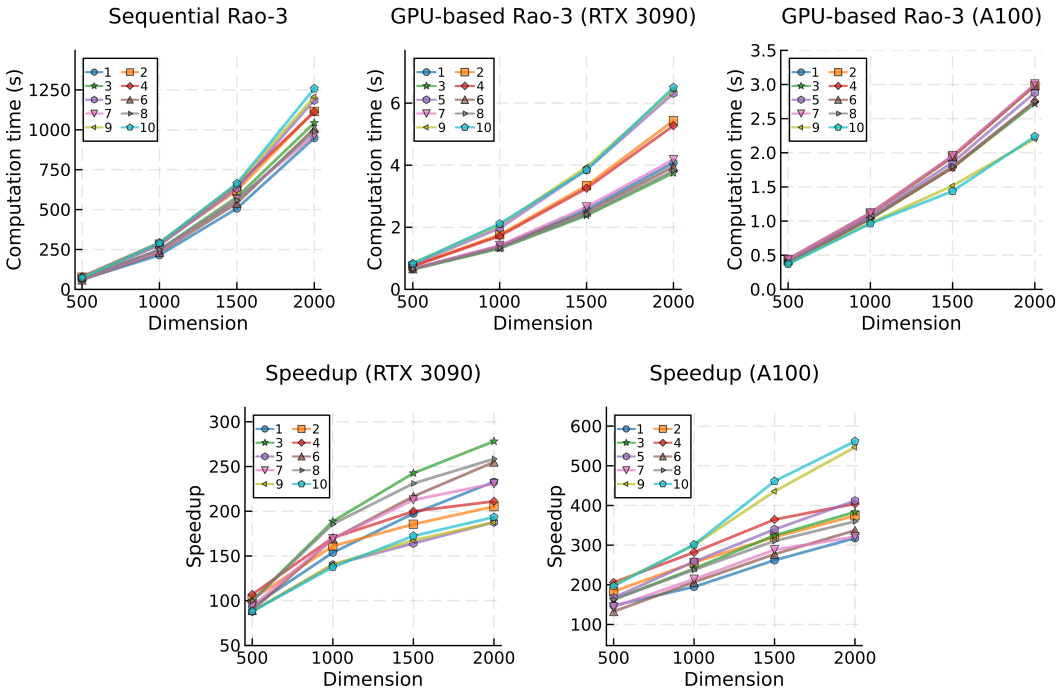

Figure 7. Mean computational times (top) and speedup (bottom) for the Rao-3 algorithm, categorized by problem and dimension.

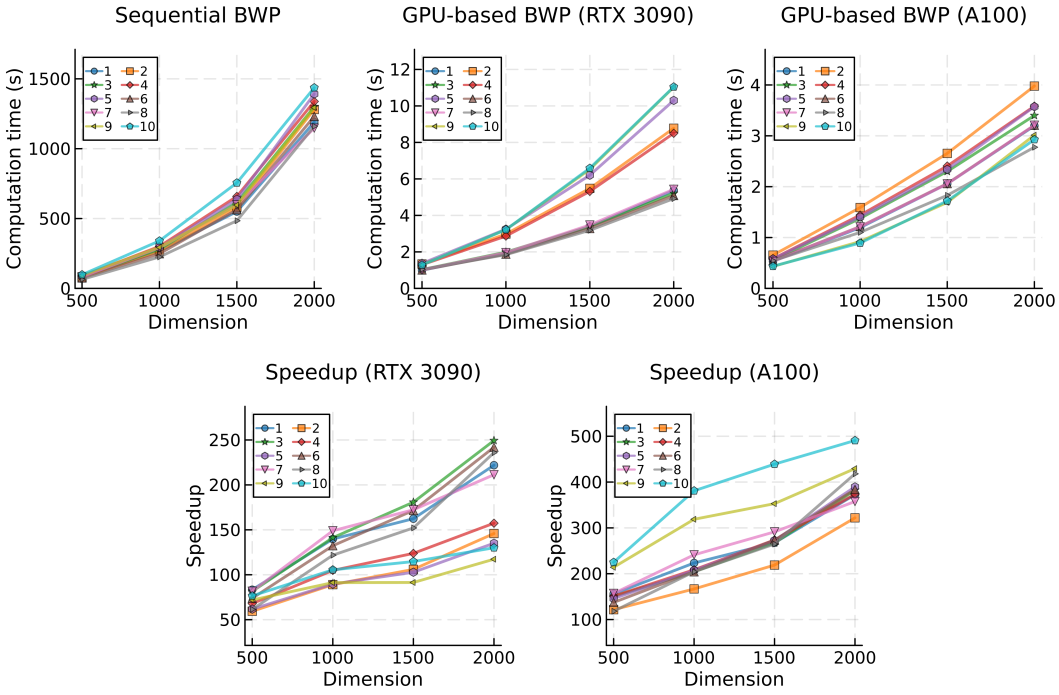

Figure 8. Mean computational times (top) and speedup (bottom) for the BWP algorithm, categorized by problem and dimension.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 1 May 2024 doi:10.20944/preprints202405.0051.v1

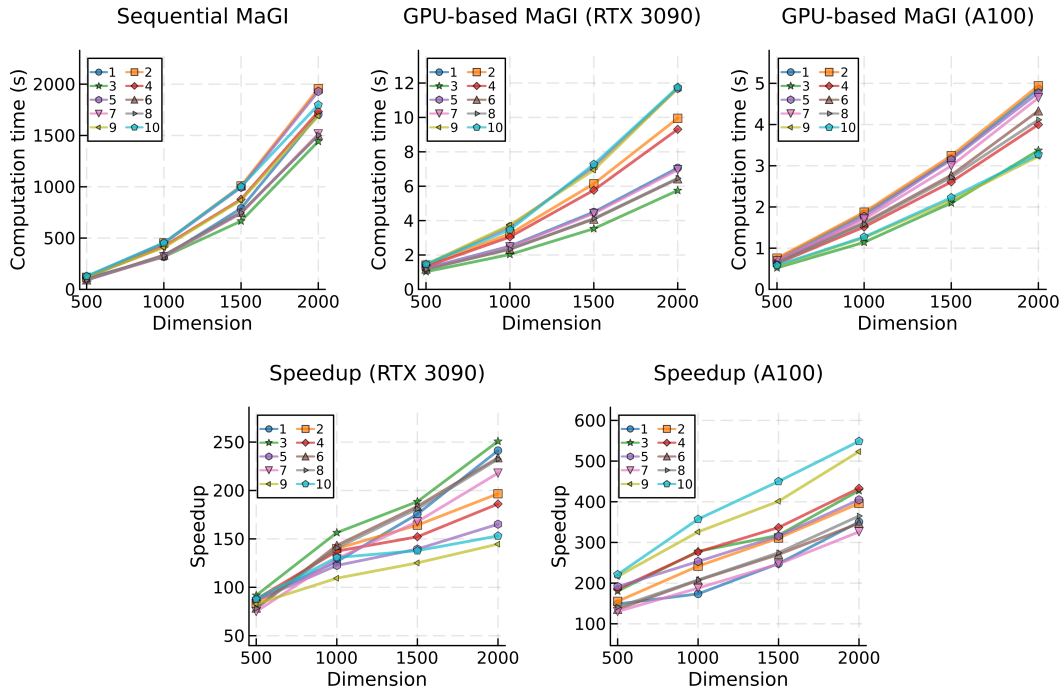

Figure 9. Mean computational times (top) and speedup (bottom) for the MaGI algorithm, categorized by problem and dimension.

Problems 1, 6, 7, and 8 usually emerge as the least computationally demanding in the sequential tests. However, no single problem consistently ranks as the fastest for all algorithms. In the context of the GPU-based implementations, tests utilizing the RTX 3090 GPU consistently position problems 3, 6, and 8 as the fastest across all tested algorithms. Nonetheless, no consistent pattern is observed when using the A100 GPU. Generally, problems 3, 8, 9, and 10 rank among the top three in terms of the lowest average execution time. When individually assessing the test problems for each algorithm, problem 8 generally appears among the least computationally intensive, in both sequential and parallel implementations, with the exception of the Rao-2 and Rao-3 algorithms, where no specific pattern has emerged.

A consistent trend observed in all sequential implementations reveals an escalating impact with the growing problem dimension, giving rise to a concave upward curve characterized by an increasing slope. This trend indicates that computational time becomes more pronounced as dimensionality increases. Such behavior aligns seamlessly with the nature of the test problems, as scaling up SNLEs generally leads to a surge in computational complexity. This underscores the imperative for efficient algorithmic strategies capable of navigating the computational challenges posed by these intricate mathematical models.

Upon analyzing the computational times for GPU-based algorithms, two key observations provide insights into the performance characteristics of the proposed parallel implementation and GPU hardware. The first observation pertains to the grouping of lines representing computation times for each test problem. The general dispersion of lines observed in the data obtained with the RTX 3090 GPU suggests that this hardware may be more influenced by problem complexity, resulting in higher variability in computational performance. In contrast, the A100 GPU exhibits a more clustered grouping of computation times, indicating more stable and predictable performance across different problem dimensions. Notably, two exceptions to this trend are observed in the GPU-based implementation of BWP (Figure 8) and MaGI (Figure 9) algorithms.

The second and more pertinent observation concerns the gradient of computation time. When testing GPU-based algorithms with the RTX 3090 GPU, increases in problem dimension result in a less pronounced rise in computational time compared to the corresponding sequential version of the same

 doi:10.20944/preprints202405.0051.v1

algorithm. Conversely, most tests conducted with the A100 GPU suggest an almost linear relationship between computational time and problem dimension. It is noteworthy, however, that some results obtained with the BWP and MaGI algorithms deviate more substantially from a linear trend. This implies that the rate at which computational time increases remains more constant with the growing problem dimension when the A100 GPU is used.

This observation suggests that GPU-based algorithms can more efficiently handle the scaling of computational demands. It implies that the parallel implementation is adept at effectively distributing and managing the workload across its parallel processing cores, thereby better controlling the rise in computational time.

Insights into the scalability and efficiency gains of the proposed parallelization strategy can be derived from an analysis of the speedup achieved during the transition from sequential to parallel GPU algorithms. Across all test scenarios, the speedup lines (bottom plots in Figures 3 through 9) exhibit a clear upward trend, indicating the proficient parallelization of the algorithms in the GPU-based implementation tested in both the RTX 3090 and A100 GPU. Furthermore, with the increase in problem dimension, the speedup consistently rises, emphasizing the effectiveness of the parallelization strategy in managing larger computational workloads and highlighting its scalability.

With the RTX 3090 GPU, variations in problem dimensions, as observed in the Jaya, BWP, and MaGI GPU-based algorithms, sometimes result in more jagged speedup lines, while the A100 GPU tends to consistently produce smoother speedup results. This observation suggests that the former GPU is more sensitive to variations in computational demands. While the speedup lines obtained with both GPUs exhibit positive growth for all algorithms across all problem dimensions, specific algorithms such as EJAYA, Rao-2, and Rao-3 display a concave downward shape in the speedups achieved with the RTX 3090, indicating a slowdown in speedup growth at higher dimensions. In contrast, speedup lines results with the A100 GPU mostly show a nearly linear or concave upward shape, signifying that the parallelization strategy effectively leverages this GPU hardware to achieve superior processing capabilities.

A noteworthy observation unfolds when examining the speedup performance across different problems. Problems 9 and 10 generally demonstrate remarkable speedup achievements with the A100 GPU. In contrast, these same problems tend to rank among the worst speedup results when executed on the RTX 3090 GPU. This disparity in speedup outcomes between the two GPUs underscores the nuanced influence of problem characteristics on the effectiveness of parallelization strategies.

A more comprehensive analysis aiming to further investigate the adaptation capabilities of the parallelization strategy across various GPU hardware configurations is presented in Table 4. The provided data includes the average times of all test problems per problem dimension and is grouped by algorithm. The computations encompass results for the sequential implementation (in the column labeled 'CPU') and the GPU-based implementation executed on four different GPUs, along with the achieved speedup gains.

The performance hierarchy among the tested GPUs indicates that the Tesla T4 exhibits the lowest overall performance, followed by the RTX 3090, Tesla V100S, and, finally, the A100, thereby showcasing the highest performance. This alignment consistently corresponds to the inherent hardware characteristics and capabilities of each GPU. The RTX 3090, distinguished by a significantly greater number of CUDA cores and higher memory bandwidth compared to the Tesla T4, naturally positions itself higher in the performance hierarchy. Despite both the Tesla V100S and A100 having fewer CUDA cores than the RTX 3090, they are intricately engineered for parallel processing, featuring significantly more advanced memory subsystems capable of facilitating data movement at substantially higher rates.

 doi:10.20944/preprints202405.0051.v1

Table 4. Mean computational results per algorithm and dimension. The time is in seconds, and the speedup gain for each GPU is shown in parentheses.

Alg.	Pop. (vars)	CPU time	Tesla T4 time (gain)	RTX 3090 time (gain)	Tesla V100S time (gain)	A100 time (gain)
Jaya	5000 (**500**)	47.25	0.82 (**57.6**)	0.62 (**75.9**)	0.36 (**132.2**)	0.31 (**153.6**)
	10 000 (**1000**)	159.50	2.72 (**58.7**)	1.33 (**120.1**)	0.80 (**199.0**)	0.74 (**216.9**)
	15 000 (**1500**)	355.90	5.84 (**61.0**)	2.44 (**145.9**)	1.44 (**247.7**)	1.24 (**286.1**)
	20 000 (**2000**)	772.46	10.16 (**76.0**)	3.84 (**201.1**)	2.28 (**338.3**)	1.86 (**415.5**)
EJAYA	5000 (**500**)	58.04	2.27 (**25.5**)	1.63 (**35.6**)	1.40 (**41.6**)	1.53 (**38.0**)
	10 000 (**1000**)	214.73	8.45 (**25.4**)	3.60 (**59.6**)	3.28 (**65.4**)	3.36 (**64.0**)
	15 000 (**1500**)	506.94	23.61 (**21.5**)	6.29 (**80.5**)	5.76 (**88.1**)	5.35 (**94.8**)
	20 000 (**2000**)	1006.64	40.25 (**25.0**)	9.85 (**102.2**)	8.85 (**113.8**)	7.54 (**133.5**)
Rao-1	5000 (**500**)	29.40	0.82 (**35.8**)	0.62 (**47.4**)	0.36 (**82.3**)	0.31 (**95.6**)
	10 000 (**1000**)	99.91	2.71 (**36.8**)	1.32 (**75.4**)	0.80 (**124.6**)	0.74 (**135.9**)
	15 000 (**1500**)	220.81	5.84 (**37.8**)	2.43 (**90.8**)	1.44 (**153.7**)	1.24 (**177.4**)
	20 000 (**2000**)	426.80	10.16 (**42.0**)	3.84 (**111.0**)	2.28 (**187.0**)	1.86 (**229.6**)
Rao-2	5000 (**500**)	59.99	1.22 (**49.2**)	0.73 (**82.2**)	0.49 (**123.3**)	0.41 (**145.7**)
	10 000 (**1000**)	232.62	4.46 (**52.1**)	1.65 (**141.4**)	1.25 (**185.6**)	1.06 (**219.6**)
	15 000 (**1500**)	536.04	10.00 (**53.6**)	3.06 (**175.2**)	2.42 (**221.7**)	1.80 (**298.1**)
	20 000 (**2000**)	1011.66	17.54 (**57.7**)	4.97 (**203.6**)	3.87 (**261.2**)	2.75 (**367.3**)
Rao-3	5000 (**500**)	68.96	1.22 (**56.6**)	0.73 (**94.6**)	0.49 (**141.7**)	0.41 (**169.2**)
	10 000 (**1000**)	261.90	4.46 (**58.7**)	1.65 (**158.9**)	1.25 (**209.5**)	1.06 (**247.7**)
	15 000 (**1500**)	597.58	10.00 (**59.8**)	3.08 (**194.3**)	2.42 (**247.3**)	1.80 (**332.7**)
	20 000 (**2000**)	1083.42	17.52 (**61.8**)	4.98 (**217.8**)	3.87 (**280.0**)	2.75 (**393.7**)
BWP	5000 (**500**)	83.07	1.59 (**52.3**)	1.17 (**71.3**)	0.62 (**133.3**)	0.54 (**154.2**)
	10 000 (**1000**)	280.36	5.48 (**51.2**)	2.51 (**111.6**)	1.42 (**197.1**)	1.23 (**227.1**)
	15 000 (**1500**)	603.98	11.95 (**50.5**)	4.68 (**128.9**)	2.63 (**229.4**)	2.11 (**286.1**)
	20 000 (**2000**)	1277.06	20.95 (**61.0**)	7.57 (**168.8**)	4.31 (**296.2**)	3.29 (**388.6**)
MaGI	5000 (**500**)	108.37	1.96 (**55.2**)	1.29 (**84.0**)	0.75 (**144.2**)	0.64 (**168.1**)
	10 000 (**1000**)	380.19	7.00 (**54.3**)	2.88 (**131.8**)	1.81 (**209.8**)	1.56 (**244.3**)
	15 000 (**1500**)	844.44	15.42 (**54.7**)	5.39 (**156.7**)	3.43 (**246.1**)	2.72 (**310.8**)
	20 000 (**2000**)	1677.47	27.02 (**62.1**)	8.70 (**192.7**)	5.66 (**296.5**)	4.15 (**403.8**)

Comparatively, the Tesla V100S and A100 GPUs differ in their configurations. The former has both a higher CUDA core count and a wider memory bus. It is important to acknowledge that critical aspects of the GPU microarchitecture, including CUDA compute capability, instruction pipeline depth, execution unit efficiency, and memory hierarchy intricacies, also contribute significantly to the overall performance of a GPU.

All tested GPUs demonstrated positive speedup ratios across all algorithms and dimensions. Nevertheless, concerning algorithms EJAYA, BWP, and MaGI, the Tesla T4 GPU exhibited a marginal decrease in average speedup ratios within some of the middle problem dimensions, followed by an increase in the highest problem dimension. Despite this marginal reduction, it is crucial to emphasize that the Tesla T4 GPU consistently maintained positive speedup ratios. This indicates a nuanced shift in the efficiency of the GPU, which could be attributed to factors related to the parallelization strategy or to GPU hardware, such as the arrangement of CUDA cores or the internal configuration of processing units.

When examining the execution times of the sequential implementations, the Rao-1 algorithm stands out as the fastest, indicative of a lower computational demand. However, this was not mirrored in the GPU results of the same algorithm. The computational times obtained with the GPU-based Rao-1, across all tested GPUs, closely resemble those acquired with the GPU-based Jaya, a mathematically

 doi:10.20944/preprints202405.0051.v1

more complex algorithm. This observation suggests a potential bottleneck in the performance of the parallelization strategy, possibly stemming from inherent characteristics of the parallelization design coupled with other factors related to GPU programming. These factors may encompass thread setup and coordination, overhead associated with kernel launch, thread synchronization mechanisms, and other CUDA programming nuances.

6. Conclusions

The analysis of GPU-based implementations of metaphor-less optimization algorithms, conducted within the framework of a standardized parallelization strategy, has yielded insightful conclusions regarding the efficacy, scalability, and performance characteristics of parallel computation across a diverse set of large-scale SNLEs with varying dimensions and tested using distinct GPU hardware configurations.

The utilization of parallel computational resources on GPUs proved highly effective in enhancing algorithmic performance. The GPU-based versions of the tested metaphor-less optimization algorithms consistently demonstrated superior efficiency compared to their sequential counterparts. In the detailed analysis conducted, the achieved speedup ranged from a minimum of 33.9× to a maximum of 561.8×. The observed speedup gains showed a proportional correlation with the augmentation of the problem dimension. Smaller dimensions resulted in more modest speedup gains, while larger dimensions led to more significant improvements. This behavior aligns with the adaptive nature of the parallelization strategy, dynamically adjusting GPU processing capabilities based on problem complexity.

The efficiency of GPU-based implementations varied among different optimization algorithms. The Rao-3 algorithm demonstrated exceptional efficiency in leveraging parallel processing capabilities, achieving the highest average speedup across all tested dimensions. Conversely, algorithms with more intricate mathematical models, such as EJAYA, exhibited more modest yet still significant speedup values. This observation underscores the impact of algorithmic complexity on the effectiveness of parallelization.

These observations were derived from results obtained with both the RTX 3090 and A100 GPUs, with the latter consistently outperforming the former despite having fewer CUDA cores. Moreover, the RTX 3090 GPU displayed higher sensitivity to variations in problem complexity, resulting in greater variability in computational performance. On the other hand, the A100 GPU showcased more stability and predictability, delivering consistent performance across different problem dimensions. This underscores the nuanced influence of GPU hardware characteristics on algorithmic efficiency.

The dynamic adaptability of the parallelization strategy to various GPU hardware configurations was further analyzed. Results revealed that all tested GPUs exhibited positive speedup ratios across all algorithms and dimensions, leading to the emergence of a performance hierarchy. The Tesla T4 demonstrated the lowest overall performance, followed by the RTX 3090, the Tesla V100S, and finally, the A100, which showcased the highest performance. This performance hierarchy is closely aligned with the intrinsic hardware characteristics of each GPU, indicating that the parallelization strategy effectively capitalized on the specific architectural features of each hardware.

Furthermore, the results highlighted that while factors such as the number of cores and memory bandwidth play a crucial role in determining GPU capabilities, they only offer a partial view of the overall performance. Such nuances underscored the importance of interpreting performance results within the context of real-world applications.

In GPU programming, the pursuit of optimal performance should be harmonized with some degrees of flexibility. While maximizing performance remains crucial, overly rigid optimized strategies may restrict algorithmic parameters or compatibility with different GPU hardware. The parallel implementation proposed in this paper showcases the capability of a generic approach to efficiently manage GPU resource allocation while providing scalability across different hardware configurations. This adaptability is achieved automatically, without requiring any source code modification, ensuring

 doi:10.20944/preprints202405.0051.v1

that applications remain responsive to evolving computational needs and effectively harness the potential of GPUs for general-purpose computing.

Refining and optimizing the parallelized metaphor-less optimization algorithms on GPU architecture, as presented in this study, unveils several promising avenues for further research. An exhaustive examination of the algorithms under consideration, with a concerted effort on refining and optimizing their parallelized versions, holds the potential for substantial enhancements to their performance. Delving deeper into the specific impact of test problems on parallel computation may yield valuable insights into the nuanced effects of parallel processing across diverse computational scenarios. Exploring dynamic parallelization strategies that adapt to varying problem complexities and GPU characteristics could also lead to additional gains in efficiency, scalability, and performance across a broader spectrum of hardware configurations. As an illustrative example, revising the method responsible for determining the block size, as suggested in this study, resulted in a noteworthy improvement of approximately 19.8% in the average performance of the GPU-based implementations.

Author Contributions: Conceptualization and validation, B.S. and L.G.L.; investigation, software, and writing—original draft preparation, B.S.; supervision, methodology, and writing—review and editing, L.G.L. Both authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Acknowledgments: The authors are grateful to Emiliano Gonçalves for providing access to the RTX 3090 GPU used in this study. The authors also acknowledge the HPC resources made available by the Portuguese National Distributed Computing Infrastructure (INCD) through the FCT (Foundation for Science and Technology) Advanced Computing Projects 2022.57951.CPCA.A0 and 2023.09611.CPCA.A1.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kelley, C.T. *Solving Nonlinear Equations with Newton's Method*; SIAM: Philadelphia, PA, 2003.
2. Pérez, R.; Lopes, V.L.R. Recent applications and numerical implementation of quasi-Newton methods for solving nonlinear systems of equations. *Numer. Alg.* **2004**, *35*, 261–285. https://doi.org/10.1023/B:NUMA.0000021762.83420.40.
3. Rice, J.R. *Numerical Methods, Software, and Analysis*, 2nd ed.; Academic Press: Boston, 1993.
4. Karr, C.; Weck, B.; Freeman, L. Solutions to systems of nonlinear equations via genetic algorithms. *Eng. Appl. Artif. Intell.* **1998**, *11*, 369–375. https://doi.org/10.1016/S0952-1976(97)00067-5.
5. Kotsireas, I.S.; Pardalos, P.M.; Semenov, A.; Trevena, W.T.; Vrahatis, M.N. Survey of methods for solving systems of nonlinear equations, Part I: Root-finding approaches, 2022. https://doi.org/10.48550/arXiv.2208.08530.
6. Li, Y.; Wei, Y.; Chu, Y. Research on solving systems of nonlinear equations based on improved PSO. *Math. Probl. Eng.* **2015**, *2015*, 1–13. https://doi.org/10.1155/2015/727218.
7. Choi, H.; Kim, S.; Shin, B.C. Choice of an initial guess for Newton's method to solve nonlinear differential equations. *Comput. Math. with Appl.* **2022**, *117*, 69–73. https://doi.org/10.1016/j.camwa.2022.04.013.
8. Press, W.J.; Teukolsky, S.A.; Vetterling, W.T.; Flannery, P.B. *Numerical Recipes in C++: The Art of Scientific Computing*, 3rd ed.; Cambridge University Press: New York, 2007.
9. Coley, D.A. *An Introduction to Genetic Algorithms for Scientists and Engineers*; World Scientific: Singapore, 1999. https://doi.org/10.1142/3904.
10. Rao, R.V. Jaya: A simple and new optimization algorithm for solving constrained and unconstrained optimization problems. *Int. J. Ind. Eng. Comput.* **2016**, *7*, 19–34. https://doi.org/10.5267/j.ijiec.2015.8.004.
11. Zhang, Y.; Chi, A.; Mirjalili, S. Enhanced Jaya algorithm: A simple but efficient optimization method for constrained engineering design problems. *Knowl. Based Syst.* **2021**, *233*, 107555. https://doi.org/10.1016/j.knosys.2021.107555.
12. Ribeiro, S.; Silva, B.; Lopes, L.G. Solving systems of nonlinear equations using Jaya and Jaya-based algorithms: A computational comparison. In Proceedings of the Proceedings of International Conference on Paradigms of Communication, Computing and Data Analytics. PCCDA 2023. Algorithms for Intelligent

Systems; Yadav, A.; Nanda, S.J.; Lim, M.H., Eds., Singapore, 2023; pp. 119–136. https://doi.org/10.1007/978-981-99-4626-6_10.

13. Rao, R.V. Rao algorithms: Three metaphor-less simple algorithms for solving optimization problems. *Int. J. Ind. Eng. Comput.* **2020**, *11*, 107–130. https://doi.org/10.5267/j.ijiec.2019.6.002.
14. Singh, R.; Gaurav, K.; Pathak, V.K.; Singh, P.; Chaudhary, H. Best–Worst–Play (BWP): A metaphor-less optimization algorithm. *J. Phys. Conf. Ser.* **2020**, *1455*, 012007. https://doi.org/10.1088/1742-6596/1455/1/012007.
15. Singh, R.; Pathak, V.K.; Srivastava, A.K.; Kumar, R.; Sharma, A. A new metaphor-less optimization algorithm for synthesis of mechanisms. *Int. J. Interact. Des. Manuf.* **2023**. https://doi.org/10.1007/s12008-023-01502-6.
16. Agushaka, J.O.; Ezugwu, A.E. Initialisation approaches for population-based metaheuristic algorithms: A comprehensive review. *Appl. Sci.* **2022**, *12*, 896. https://doi.org/10.3390/app12020896.
17. Agushaka, J.O.; Ezugwu, A.E. Advanced arithmetic optimization algorithm for solving mechanical engineering design problems. *PLOS ONE* **2021**, *16*, 1–29. https://doi.org/10.1371/journal.pone.0255703.
18. Freitas, D.; Lopes, L.G.; Morgado-Dias, F. Particle swarm optimisation: A historical review up to the current developments. *Entropy* **2020**, *22*, 362. https://doi.org/10.3390/e22030362.
19. Rao, R.V. *Jaya: An Advanced Optimization Algorithm and its Engineering Applications*; Springer: Cham, Switzerland, 2019.
20. Zitar, R.A.; Al-Betar, M.A.; Awadallah, M.A.; Doush, I.A.; Assaleh, K. An intensive and comprehensive overview of JAYA algorithm, its versions and applications. *Arch. Comput. Methods Eng.* **2022**, *29*, 763–792. https://doi.org/10.1007/s11831-021-09585-8.
21. Civicioglu, P. Backtracking search optimization algorithm for numerical optimization problems. *Appl. Math. Comput.* **2013**, *219*, 8121–8144. https://doi.org/10.1016/j.amc.2013.02.017.
22. Soyata, T. *GPU Parallel Program Development Using CUDA*; Taylor and Francis: Boca Raton, FL, 2018.
23. Gogolińska, A.; Mikulski, Ł.; Piątkowski, M. GPU computations and memory access model based on Petri nets. In *Transactions on Petri Nets and Other Models of Concurrency XIII*; Koutny, M.; Kristensen, L.; Penczek, W., Eds.; Springer: Berlin, 2018; Vol. 11 090, *Lecture Notes in Computer Science*, pp. 136–157. https://doi.org/10.1007/978-3-662-58381-4_7.
24. Cheng, J.R.; Gen, M. Parallel genetic algorithms with GPU computing. In *Industry 4.0 – Impact on Intelligent Logistics and Manufacturing*; Bányai, T.; Petrillo, A.; De Felice, F., Eds.; IntechOpen: Rijeka, Croatia, 2020; pp. 69–93. https://doi.org/10.5772/intechopen.89152.
25. Wang, L.; Zhang, Z.; Huang, C.; Tsui, K.L. A GPU-accelerated parallel Jaya algorithm for efficiently estimating Li-ion battery model parameters. *Appl. Soft Comput.* **2018**, *65*, 12–20. https://doi.org/10.1016/j.asoc.2017.12.041.
26. Jimeno-Morenilla, A.; Sánchez-Romero, J.L.; Migallón, H.; Mora-Mora, H. Jaya optimization algorithm with GPU acceleration. *J. Supercomput.* **2019**, *75*, 1094–1106. https://doi.org/10.1007/s11227-018-2316-7.
27. Silva, B.; Lopes, L.G., An efficient GPU parallelization of the Jaya optimization algorithm and its application for solving large systems of nonlinear equations. In *Book of Abstracts of the III International Conference on Optimization, Learning Algorithms and Applications – OL2A 2023, Ponta Delgada, Azores, Portugal, September 27–29, 2023*; Pereira, A.I.; Mendes, A.; Fernandes, F.P.; Coelho, J.P.; Teixeira, J.P.; Cascalho, J.; Lima, J.; Pacheco, M.F.; Lopes, R.P., Eds.; IPB: Bragança, Portugal, 2023; p. 87.
28. Silva, B.; Lopes, L.G., An efficient GPU parallelization of the Jaya optimization algorithm and its application for solving large systems of nonlinear equations. In *Optimization, Learning Algorithms and Applications: Third International Conference, OL2A 2023, Ponta Delgada, Portugal, September 27–29, 2023, Revised Selected Papers, Part II*; Pereira, A.I.; Mendes, A.; Fernandes, F.P.; Pacheco, M.F.; Coelho, J.P.; Lima, J., Eds.; CCIS, vol. 1982, Springer: Cham, Switzerland, 2024; pp. 368–381. https://doi.org/10.1007/978-3-031-53036-4_26.
29. Silva, B.; Lopes, L.G., GPU acceleration of the Enhanced Jaya optimization algorithm for solving large systems of nonlinear equations. In *Book of Abstracts of the 4th International Conference on Numerical Computations: Theory and Algorithms – NUMTA 2023, Pizzo, Calabria, Italy, 14–20 June 2023*; Sergeyev, Y.D.; Kvasov, D.E.; Nasso, M.C., Eds.; Università della Calabria, DIMES: Rende (CS), Italy, 2023; p. 190.
30. Silva, B.; Lopes, L.G., GPU acceleration of the Enhanced Jaya optimization algorithm for solving large systems of nonlinear equations. In *Numerical Computations: Theory and Algorithms (NUMTA 2023)*; Sergeyev, Y.D.; Kvasov, D.E.; Astorino, A., Eds.; Lecture Notes in Computer Science, Springer: Cham, Switzerland, 2024 (to appear).

31. Silva, B.; Lopes, L.G., GPU-based acceleration of the Rao optimization algorithms: Application to the solution of large systems of nonlinear equations. In *Intelligent Data Engineering and Automated Learning – IDEAL 2023*; Quaresma, P.; Camacho, D.; Yin, H.; Gonçalves, T.; Julian, V.; Tallón-Ballesteros, A.J., Eds.; LNCS, vol. 14404, Springer: Cham, Switzerland, 2023; pp. 107–119. https://doi.org/10.1007/978-3-031-48232-8_11.
32. Silva, B.; Lopes, L.G., A massively parallel BWP algorithm for solving large-scale systems of nonlinear equations. In *2023 IEEE High Performance Extreme Computing Conference (HPEC), Boston, MA, USA, 2023*; 2023; pp. 1–6. https://doi.org/10.1109/HPEC58863.2023.10363575.
33. Bezanson, J.; Edelman, A.; Karpinski, S.; Shah, V.B. Julia: A fresh approach to numerical computing. *SIAM Rev.* **2017**, *59*, 65–98. https://doi.org/10.1137/141000671.
34. Besard, T.; Foket, C.; De Sutter, B. Effective extensible programming: Unleashing Julia on GPUs. *IEEE Trans. Parallel Distrib. Syst.* **2019**, *30*, 827–841. https://doi.org/10.1109/TPDS.2018.2872064.
35. Gao, K.; Mei, G.; Piccialli, F.; Cuomo, S.; Tu, J.; Huo, Z. Julia language in machine learning: Algorithms, applications, and open issues. *Comput. Sci. Rev.* **2020**, *37*, 100254. https://doi.org/j.cosrev.2020.100254.
36. Etiemble, D. 45-year CPU evolution: one law and two equations. In Proceedings of the Second Workshop on Pioneering Processor Paradigms, Vienne, Austria, 2018. https://doi.org/10.48550/arXiv.1803.00254.
37. Moré, J.J.; Garbow, B.S.; Hillstrom, K.E. Testing unconstrained optimization software. *ACM Trans. Math. Softw.* **1981**, *7*, 17–41. https://doi.org/10.1145/355934.355936.
38. Friedlander, A.; Gomes-Ruggiero, M.A.; Kozakevich, D.N.; Martínez, J.M.; Santos, S.A. Solving nonlinear systems of equations by means of quasi-Newton methods with a nonmonotone strategy. *Optim. Methods Softw.* **1997**, *8*, 25–51. https://doi.org/10.1080/10556789708805664.
39. Bodon, E.; Del Popolo, A.; Lukšan, L.; Spedicato, E. Numerical performance of ABS codes for systems of nonlinear equations. Technical Report DMSIA 01/2001, Universitá degli Studi di Bergamo, Bergamo, Italy, 2001.
40. Ziani, M.; Guyomarc'h, F. An autoadaptative limited memory Broyden's method to solve systems of nonlinear equations. *Appl. Math. Comput.* **2008**, *205*, 202–211. https://doi.org/10.1016/j.amc.2008.06.047.
41. Kelley, C.T.; Qi, L.; Tong, X.; Yin, H. Finding a stable solution of a system of nonlinear equations. *J. Ind. Manag. Optim.* **2011**, *7*, 497–521. https://doi.org/10.3934/jimo.2011.7.497.