GPU-Based Parallelization of Differential Evolution Variants: Application to Solving Large Systems of Nonlinear Equations

Diogo Martins
Master's Program in Computer Eng.
Faculty of Exact Sci. and Eng.
University of Madeira
Funchal, Madeira, Portugal
2082921@student.uma.pt

Bruno Silva
Reg. Secr. of Educ., Sci. and Technol.
Regional Government of Madeira
FCEE, University of Madeira
Funchal, Madeira, Portugal
bruno.silva@madeira.gov.pt

Luiz Guerreiro Lopes
Fac. of Exact Sciences and Engineering
University of Madeira
Funchal, Madeira, Portugal
NOVA LINCS, Caparica, Portugal
lopes@uma.pt

***Abstract*—Differential Evolution (DE) is a well-established evolutionary algorithm widely used for global optimization over continuous search spaces. However, it can become computationally demanding when applied to large-scale optimization problems. Modern GPUs offer massive parallelism, high memory bandwidth, and high floating-point throughput, making them well-suited to accelerating population-based algorithms. This paper presents GPU-accelerated implementations of the standard DE algorithm and three well-known variants: SHADE, jDE, and CoDE. Their computational performance and scalability are evaluated using ten large-scale systems of nonlinear equations, reformulated as nonlinear optimization problems, with dimensionalities ranging from 500 to 2,000 variables. Execution times obtained on the NVIDIA A100 and V100S GPUs are compared with those of the corresponding sequential implementation running on an AMD EPYC 7643 CPU. The results show substantial reductions in execution time for all four algorithms and across all tested problem dimensions, with mean speedups reaching approximately 69× on the A100 and 42× on the V100S. The A100 also exhibits better scalability at the largest problem dimensions.**

***Index Terms*—metaheuristic optimization, differential evolution, parallel evolutionary algorithms, GPU acceleration, large-scale optimization, nonlinear equation systems**

I. Introduction

Evolutionary Algorithms (EAs) are stochastic, population-based algorithms inspired by the principles of Darwin's theory of evolution. They explore different regions of the search space to efficiently guide convergence, making them particularly effective for solving complex optimization problems [1].

Differential Evolution (DE), proposed in 1997 by Storn and Price [2], is one such EA, well-established for its simplicity and competitive performance across a broad range of optimization tasks. However, as problem dimensionality and complexity increase, DE can become computationally demanding.

GPU computing provides powerful parallel computational capabilities, enabling the concurrent execution of thousands of threads. In addition, GPUs offer significantly higher memory bandwidth and floating-point computational throughput than CPUs [3], making them an ideal platform for accelerating large-scale optimization algorithms.

This paper investigates GPU-accelerated implementations of the standard DE algorithm and three of its variants, namely SHADE, jDE, and CoDE, by parallelizing all evolutionary operators, thereby exploiting the massive parallel architecture of modern GPUs. The resulting GPU-based implementations are assessed in terms of the speedup achieved relative to their sequential counterparts.

To evaluate the proposed implementations under demanding and realistic conditions, a set of large-scale Systems of Nonlinear Equations (SNLEs) is employed as a benchmark application. It should be noted that DE is a general-purpose metaheuristic optimization algorithm, not one designed specifically for solving SNLEs; these systems are used here solely as a challenging test domain, since, due to their nonlinear nature, they can be computationally expensive to solve when reformulated as nonlinear optimization problems, and they represent real-world scenarios widely encountered across several scientific fields.

This work focuses on evaluating the computational performance and high-dimensional scalability of DE variants on massively parallel GPU architectures for solving SNLEs, rather than on a systematic cross-hardware comparison with optimized multicore CPU implementations.

II. Differential Evolution algorithm and Variants

Although having vector-based operators rather than being fully inspired by biological evolution, DE is still classified as an EA, evolving a population of candidates through mutation, crossover, and selection. Initially, DE randomly generates a population of NP candidates within the initially defined bounds, where each candidate is a D-dimensional vector.

DE perturbs the population to generate new candidates, iteratively evolving the population toward increasingly competitive solutions until a stopping criterion is met. To create a new candidate, the mutation operator is applied first. For

each target vector, x_i, a mutant vector, v_i, is generated by applying weighted differences between selected population vectors, scaled by the scale factor (F). The scale factor F controls the magnitude of the differential mutation [4].

The mutation operator follows a predefined mutation strategy that determines how v_i is calculated. Every strategy follows the convention DE/x/y/z; where x specifies the base vector, y stands for the number of difference vectors, and z indicates the crossover scheme used [1].

Then, the trial vector, u_i, is generated by combining v_i with x_i, using the binomial crossover. For each dimension $j \in \{1, \ldots, D\}$, the j-th component of u_i has probability CR of being inherited from v_i; otherwise, it is inherited from x_i:

$$u_{ij} = \begin{cases} v_{ij}, & \text{if } \mathrm{rand}_j(0,1) < CR \text{ or } j = j_{\mathrm{rand}} \\ x_{ij}, & \text{otherwise.} \end{cases} \tag{1}$$

The j_{rand} component is added to ensure that at least one component is copied from the mutant, so that u_i is not equal to x_i.

Once u_i is created, a greedy selection method is applied. The objective function values of u_i and x_i are compared, and the vector with the lower value is retained:

$$x_i^{(g+1)} = \begin{cases} u_i, & \text{if } f(u_i) \leq f(x_i) \\ x_i, & \text{otherwise.} \end{cases} \tag{2}$$

A. SHADE

SHADE [5] is a self-adaptive parameter control variant of the DE algorithm, proposed to address the parameter setting problem. Control parameter tuning is problem-dependent and highly influences performance.

SHADE follows the mutation strategy DE/current-to-pbest/1. This strategy was presented in [6] to improve the mutation scheme DE/current-to-best/1, by adding an external archive, A, and a parameter p to adjust the greediness. The resulting mutant vector is given by

$$v_i = x_i + F_i \cdot (x_{pbest} - x_i) + F_i \cdot (x_{r1} - x_{r2}). \tag{3}$$

SHADE maintains memory arrays for the control parameters F and CR, M_F, M_{CR}, each of size H. Both arrays are initialized to 0.5 for all entries:

$$M_{F_i} = M_{CR_i} = 0.5, \quad \forall i \in \{1, \ldots, H\}. \tag{4}$$

During the optimization process, each candidate evaluation starts by generating the control parameters. An entry from M_F and M_{CR} is chosen randomly, and then the F and CR parameters are calculated by sampling from Cauchy and normal distributions, respectively.

If the candidate evaluation results in an improvement, the corresponding control parameters are stored in the temporary sets S_F and S_{CR}. At the end of each generation, M_F and M_{CR} are updated using the weighted Lehmer mean, W_L, and the weighted arithmetic mean, W_A, respectively, as shown in (5) and (6), respectively [7].

$$M_{F,k,G+1} = \begin{cases} \mathrm{mean}_{W_L}(S_F) & \text{if } S_F \neq \varnothing \\ M_{F,k,G} & \text{otherwise} \end{cases} \tag{5}$$

$$M_{CR,k,G+1} = \begin{cases} \mathrm{mean}_{W_A}(S_{CR}) & \text{if } S_{CR} \neq \varnothing \\ M_{CR,k,G} & \text{otherwise} \end{cases} \tag{6}$$

The position to be updated is set by index k; at each generation this value is incremented by 1 until it reaches H, after which it is reset to 1.

B. jDE

Self-adaptive DE, jDE, was presented in [8] to address the parameter tuning problem, noting that this process may involve a trial-and-error approach. The control parameters are encoded within the population by extending each individual with its own F and CR values.

Before generating the trial vector for each individual, the associated control parameters are updated as follows:

$$F_{i,G+1} = \begin{cases} F_l + \mathrm{rand}_1 F_u, & \text{if } \mathrm{rand}_2 < \tau_1, \\ F_{i,G}, & \text{otherwise,} \end{cases}$$

$$CR_{i,G+1} = \begin{cases} \mathrm{rand}_3, & \text{if } \mathrm{rand}_4 < \tau_2, \\ CR_{i,G}, & \text{otherwise.} \end{cases}$$

The variables τ_1 and τ_2 define the probabilities of updating F and CR, respectively. The constants $F_l = 0.1$ and $F_u = 0.9$ result in newly generated values of F in the interval $[0.1, 1.0]$.

C. CoDE

Rather than being a self-adaptive parameter control variant like the previous ones, the CoDE variant [9] employs several predefined combinations of control parameters and trial generation strategies.

Before each candidate evaluation, a parameter control combination is randomly selected in order to proceed with the trials generation. The parameter combinations adopted are the following:

1) $[F = 1.0, CR = 0.1]$
2) $[F = 1.0, CR = 0.9]$
3) $[F = 0.8, CR = 0.2]$

CoDE employs different trial generation strategies, each expected to offer distinct advantages during different phases of convergence. These strategies are based on the existing literature and are the following:

1) DE/rand/1/bin:

$$v_i = x_{r1} + F \cdot (x_{r2} - x_{r3}) \tag{7}$$

2) DE/rand/2/bin:

$$v_i = x_{r1} + F \cdot (x_{r2} - x_{r3} + x_{r4} - x_{r5}) \tag{8}$$

3) current-to-rand/1:

$$v_i = x_i + rand \cdot (x_{r1} - x_i) + F \cdot (x_{r2} - x_{r3}) \tag{9}$$

Note that (9) does not specify the type of crossover used. The reason is that the trial generated by this strategy does not go through a crossover phase.

III. GPU Parallelization of Differential Evolution and Variants

A. CUDA programming

CUDA (Compute Unified Device Architecture) is a platform developed by NVIDIA that extends the C/C++ programming languages with abstractions for parallel execution. It enables full exploitation of GPU capabilities for general-purpose parallel computing. However, achieving high performance requires a full understanding of the GPU architecture, as well as a careful analysis of data structures [10].

In CUDA, each block inside the grid has a unique block index, and all threads within a block are identified by a unique thread index. Depending on the characteristics of the algorithm and the data layout, thread blocks can be organized in one, two, or three dimensions. This provides flexibility to efficiently map the data onto threads and optimize memory access patterns in order to maximize parallelism [10].

CUDA provides several memory spaces organized in a hierarchical structure, as shown in Fig. 1. Efficient use of these memories is essential for reducing memory latency and, therefore, improving kernel performance.

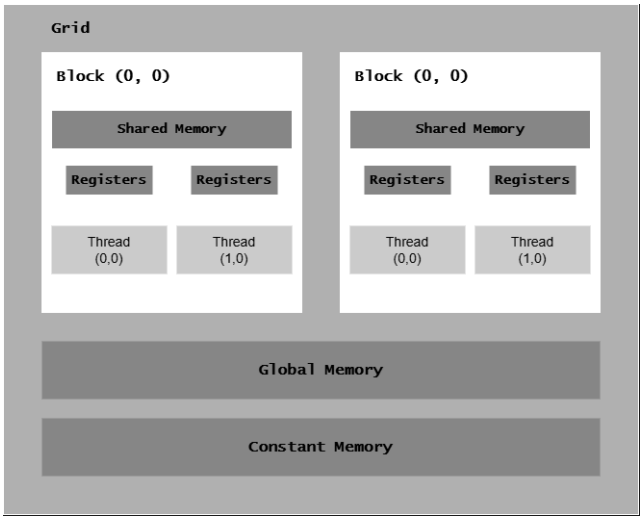


Fig. 1: CUDA device memory model.

B. CUDA Implementation of DE and Variants

The population is organized as a NPxD CUDA array in column-major order. The CUDA kernels were designed to assign each thread to one individual and iterate over all dimensions. This choice promotes coalesced global memory accesses, since accesses to the same dimension by consecutive threads are adjacent in memory. However, the mutation operator requires access to randomly selected individuals for the trial vector generation, resulting in non-coalesced memory accesses during this phase.

The generation of the trial vector requires consistency in the values used throughout the process, particularly the randomly generated indices for the mutation operator and the j_{rand} value used during the crossover phase. These values must remain consistent for each candidate, which justifies mapping one thread to each individual.

In the selection phase, in case of improvement, the trial vector needs to be copied to the corresponding position in the population dimension by dimension, since array slicing is not supported within CUDA kernels. In this case, the one-thread-per-individual mapping was retained, allowing each thread to evaluate its trial vector, and, if an improvement occurs, update the population. The fitness evaluation and selection phases are combined into one kernel, avoiding extra kernel launches.

Efficiently managing hardware resources depends on the GPU architecture. For this reason, the number of threads per block is dynamically determined according to the maximum number of threads per block supported by the current GPU, denoted by NT. This value is then used to determine the number of blocks required to process the population, according to:

$$N_{\text{blocks}} = \left\lceil \frac{NP}{NT} \right\rceil \tag{10}$$

The data structures were designed to remain persistent on the device. The population, the trial vectors and the objective values were allocated directly on the GPU using CuArrays. This avoids unnecessary data transfers between host and device during the evolutionary process, reducing the associated data transfer overhead.

Algorithm 1 GPU-based DE

1: Initialize population D, NP, G_{max}
2: $P \leftarrow kernel_init_pop_fit_eval()$
3: $i \leftarrow 0$
4: **while** $i < G_{max}$ **do**
5: $\quad trials \leftarrow kernel_trial_generator(P)$
6: $\quad P_i \leftarrow kernel_selection(P, trials, fx())$
7: $\quad i \leftarrow i + 1$
8: **end while**
9: **Return** best solution

The main cycle of the DE is divided into two stages: in the first stage, a kernel generates the trial vectors by performing the mutation and crossover operators for each candidate solution. In the second stage, a kernel is responsible for performing the selection phase.

Grouping different operators within the same kernel can reduce launch overhead and decrease the number of global memory accesses. However, this approach may increase register pressure. On the other hand, separating operators into different kernels facilitates the management of thread synchronization and dependencies. Therefore, this approach reflects a trade-off between maintaining efficient thread synchronization and reducing kernel launch overhead.

In the SHADE algorithm, the parameter calculation is particularly challenging. This variant stores the parameters that resulted in improvements and recalculates them at the end of each generation. This mechanism introduces dependencies that

limit the number of threads and blocks needed to calculate the new parameters.

Algorithm 2 SHADE selection kernel

```
function kernel_selection(P, trials, fx())
    row = (blockIdx.x − 1) · blockDim.x + threadIdx.x
    if row ≤ NP then
        fit ← fx(trials[row])
        if fit < fitness[row] then
            fits[row] ← fit
            for col ← 1 to D do
                P[row, col] ← trials[col, row]
            end for
            map[row] = 1
        else
            map[row] = 0
        end if
    end if
end function
```

The kernel responsible for the selection operator, illustrated in Algorithm 2, records, in a map array, which individuals in the population improved. Subsequently, a separate kernel uses this map to store the parameters associated with the successful evaluations. The number of threads launched by this kernel corresponds to the total number of improvements, calculated by summing the entries in the map array.

At the end of each generation, the adopted approach calculates the new parameters on the host using the weighted Lehmer and weighted arithmetic means described in Section II-A; the updated parameters are then copied to global memory. The computational workload associated with this phase is insufficient to justify launching an additional kernel, as it would result in performance degradation.

The jDE variant, unlike SHADE, does not require maintaining historical memory arrays for adapting the control parameters. The GPU-based jDE implementation follows a structure similar to Algorithm 1. Each individual maintains its own control parameters, and the kernel responsible for trial generation also updates the corresponding control parameters for each candidate as described in Section II-B.

In the CoDE variant, the three trial vectors are generated in the same kernel. However, this design results in more work performed per thread. To compensate for the increased computational workload and increased register usage, the number of threads per block was reduced. A design where the trial vectors were generated in separate kernels was also tested and resulted in longer execution times. In this case, reducing the number of threads per block to compensate for the increased computational workload per thread resulted in better performance than a more balanced workload distributed across multiple thread executions.

IV. Experimental Setup

To evaluate the efficiency of the proposed GPU-based implementations in real-world scenarios, the experiments were carried out on high-dimensional SNLEs. Specifically, the evaluations scale across a high-dimensional problem space where $D \in \{500, 1000, 1500, 2000\}$, with a population size set to $10D$. The results will be compared with a sequential implementation.

TABLE I: Benchmark problems.

ID	Benchmark problem	Domain and parameters
P1	Broyden tridiagonal function [12]	$[-1, 1]^D$
P2	Discrete boundary value function [12]	$[0, 5]^D,\ t_i = ih$
P3	Modified Rosenbrock function [13]	$[-10, 10]^D$
P4	Powell badly scaled function [13]	$[0, 100]^D$
P5	Beam problem [14]	$[-100, 100]^D,\ \alpha = 11$
P6	Bratu's problem [14]	$[-100, 100]^D,\ \alpha = 3.5$
P7	Extended Rosenbrock function [12]	$[-100, 100]^D$
P8	Schubert–Broyden function [15]	$[-100, 100]^D$
P9	Extended Powell singular function [12]	$[-5, 5]^D$
P10	Martínez function [16]	$[-100, 100]^D$

$[a, b]^D$ denotes the D-dimensional box $[a, b] \times \cdots \times [a, b]$.
For P2, P5, and P6, $h = (n + 1)^{-1}$.

The set of SNLEs selected to evaluate the accelerated implementations is representative of several scientific domains [11]. Specifically, ten benchmark problems were chosen for their scalability and the distinct computational challenges they present. These problems are summarized in Table I.

The sequential and GPU-based implementations use different random number generation (RNG) mechanisms. The CPU implementation uses Julia's standard pseudorandom number generator, whereas random numbers within the GPU kernels are generated using CUDA.jl's device-side RNG. Consequently, even when initialized with the same seed, the two implementations are not expected to generate identical random sequences or bitwise-identical optimization trajectories. Each experiment consisted of 31 independent runs to reduce sensitivity to stochastic variability and provide statistically robust performance estimates.

Both sequential and GPU-based implementations use double precision arithmetic. The population, trial vectors and objective values are explicitly allocated as Float64 arrays in the CPU implementation and as Float64 CUDA arrays in the GPU implementation. Additionally, the random floating point numbers generated using julia.rand() are Float64 values, while random numbers generated within CUDA kernels are explicitly specified as rand(Float64) to ensure double precision arithmetic.

End-to-end execution time was used as the primary performance metric, and speedup was computed relative to the sequential CPU implementation using mean execution times. Different thread-block configurations and kernel organizations were also explored during implementation to determine suitable configurations for the evaluated workloads.

All experiments were implemented in Julia [17], version 1.11.4, using CUDA.jl for GPU kernel programming [18]. The sequential execution was performed on an AMD EPYC 7643 CPU running at 2.3 GHz with a boost of 3.6 GHz, featuring 48 cores and 96 threads. The GPU parallel implementation

was executed on an NVIDIA A100 PCIe GPU with 6,912 CUDA cores and 80 GB of VRAM, and on an NVIDIA Tesla V100S PCIe with 5,120 CUDA cores and 32 GB of VRAM.

V. Results and Discussion

The results show a substantial reduction in execution times for both GPUs compared with the CPU across all algorithms and problem dimensionalities, indicating that GPU-based implementations outperform the corresponding sequential implementations for this class of metaheuristic optimization algorithms. Table II presents the average execution times obtained on the CPU and GPUs, together with the corresponding speedup factors.

Across all experiments, the DE algorithm and the CoDE variant showed the highest speedup values. However, the CoDE and SHADE variants presented the highest execution times on the GPUs. This behavior is associated with the greater computational complexity of both variants. CoDE requires the generation of multiple trial vectors per generation, thereby increasing the computational workload, whereas SHADE requires the storage and adaptation of control parameters, which introduce additional dependencies and data transfers between the host and device.

Problems 9 and 10 exhibited higher execution times in the sequential implementations than the remaining problems, indicating that they impose an additional computational cost on the CPU. However, in the GPU-based implementations, the execution times across the benchmark problems were more balanced, and this behavior was not observed. This suggests that the GPU is more efficient than the CPU at handling the additional computational workload associated with more complex problems.

Figures 2 and 3 show the evolution of the mean speedup obtained on the NVIDIA A100 and NVIDIA V100S, respectively, across the tested problem dimensionalities. The plotted values were calculated by averaging the speedup obtained for all benchmark problems. Although speedup is problem-dependent, these figures illustrate the overall evolution of the speedup achieved on both GPUs.

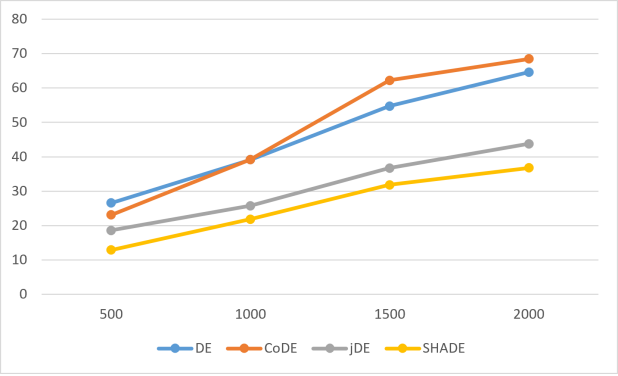


Fig. 2: Mean speedup by dimension on the NVIDIA A100

The results demonstrate the superior performance of the NVIDIA A100 compared with the NVIDIA V100S. For the smaller problem dimensionalities, the execution times did not show a clear performance advantage for either GPU, as both achieved similar results. However, at $D = 1500$, the speedup obtained on the NVIDIA V100S showed little improvement compared with that obtained at $D = 1000$. At $D = 2000$, the speedup on the NVIDIA V100S decreased compared with that obtained at $D = 1500$. In contrast, the NVIDIA A100 continued to achieve increasing speedups across all tested problem dimensionalities, showing that it handles the increase in computational workload more effectively and provides better scalability than the NVIDIA V100S.

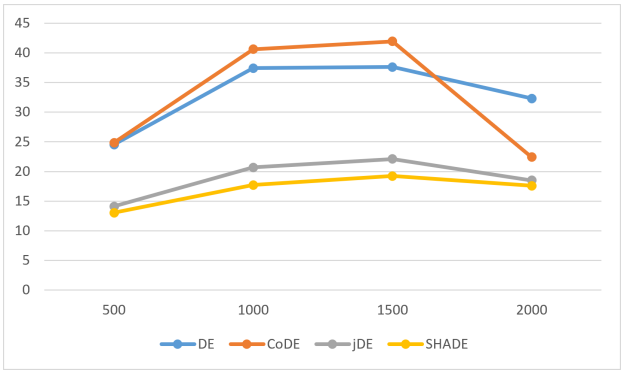


Fig. 3: Mean speedup by dimension on the NVIDIA V100S

The CoDE variant proved to be more computationally demanding than the remaining algorithms, particularly as the problem dimensionality rises. When executed on the NVIDIA V100S, its speedup decreased sharply, indicating that the additional computational workload imposed by this algorithm limited its scalability on this GPU.

VI. Conclusion

This study proposed GPU-based parallel implementations of the standard DE algorithm and the SHADE, jDE, and CoDE variants, analyzing their performance improvements and scalability. The experimental results demonstrate that these GPU-based implementations can significantly reduce execution times compared with their sequential counterparts, demonstrating the potential of GPU parallelization for accelerating this class of optimization algorithms.

Specifically, this work demonstrates that GPU parallelization yields substantial performance gains, achieving speedups of up to $108\times$ over the sequential CPU implementation. To build upon these findings, this work can be extended to include a deeper investigation into alternative parallelization strategies, specialized kernel architectures, and memory access patterns to further optimize execution throughput and scalability.

Acknowledgment

This work is supported by NOVA LINCS (UID/04516/2025) with the financial support of FCT.IP (DOI: 10.54499/UID/04516/2025). The HPC resources used in this study were made available by the Portuguese National Distributed Computing Infrastructure / National Center for

TABLE II: Comparison of average CPU and GPU execution times (in seconds) and speedup, for all dimensions ($D \in \{500, 1000, 1500, 2000\}$).

		DE						**CoDE**						**jDE**						**SHADE**					
		V100S			**A100**			**V100S**			**A100**			**V100S**			**A100**			**V100S**			**A100**		
Dim.	Prob.	CPU	GPU	speedup	CPU	GPU	speedup	CPU	GPU	speedup	CPU	GPU	speedup	CPU	GPU	speedup	CPU	GPU	speedup	CPU	GPU	speedup	CPU	GPU	speedup
500	P1	28.42	1.51	18.86	28.42	1.38	20.61	49.25	2.73	18.03	49.25	2.95	16.69	23.67	2.18	10.88	23.67	1.66	14.25	29.83	2.83	10.54	29.83	2.83	10.55
	P2	30.30	1.53	19.86	30.30	1.34	22.58	54.29	2.73	19.87	54.29	2.97	18.27	25.17	2.16	11.65	25.17	1.67	15.06	31.02	2.84	10.90	31.02	2.82	11.00
	P3	30.27	1.31	23.15	30.27	1.23	24.64	51.72	2.40	21.56	51.72	2.63	19.69	24.78	2.00	12.38	24.78	1.53	16.24	30.81	2.66	11.58	30.81	2.63	11.70
	P4	37.07	1.55	23.86	37.07	1.43	25.87	71.66	2.84	25.24	71.66	3.05	23.49	32.07	2.24	14.35	32.07	1.70	18.89	39.82	2.88	13.81	39.82	2.89	13.78
	P5	44.12	1.40	31.53	44.12	1.28	34.55	84.61	2.59	32.63	84.61	2.74	30.90	37.07	2.08	17.84	37.07	1.52	24.34	45.80	2.44	18.81	45.80	2.60	17.65
	P6	29.14	1.49	19.51	29.14	1.36	21.35	45.25	2.55	17.72	45.25	2.80	16.18	26.00	2.18	11.93	26.00	1.68	15.50	29.37	2.84	10.34	29.37	2.82	10.42
	P7	28.00	1.47	19.02	28.00	1.36	20.57	52.58	2.58	20.41	52.58	2.81	18.73	24.75	2.16	11.44	24.75	1.65	15.03	29.50	2.81	10.50	29.50	2.83	10.44
	P8	28.73	1.38	20.76	28.73	1.25	23.08	51.62	2.46	20.96	51.62	2.63	19.65	25.54	2.07	12.36	25.54	1.52	16.85	30.67	2.72	11.29	30.67	2.70	11.37
	P9	45.54	1.34	33.98	45.54	1.27	35.84	95.12	2.65	35.87	95.12	2.79	34.07	42.80	2.29	18.70	42.80	1.71	25.03	47.97	2.95	16.27	47.97	2.95	16.25
	P10	46.69	1.34	34.76	46.69	1.27	36.65	96.30	2.65	36.29	96.30	2.89	33.37	43.11	2.19	19.70	43.11	1.73	24.93	47.87	2.96	16.19	47.87	2.96	16.19
1000	P1	99.07	3.55	27.88	99.07	3.39	29.21	163.66	5.80	28.23	163.66	6.28	26.06	80.54	5.32	15.15	80.54	4.28	18.81	104.92	7.40	14.19	104.92	5.93	17.69
	P2	103.49	3.59	28.80	103.49	3.36	30.83	178.67	5.77	30.96	178.67	5.32	33.58	89.25	5.31	16.81	89.25	4.31	20.72	107.94	7.75	13.93	107.94	5.97	18.08
	P3	101.67	3.18	31.99	101.67	2.99	34.01	167.62	5.12	32.72	167.62	5.32	31.50	86.62	4.99	17.37	86.62	3.85	22.48	106.52	7.30	14.59	106.52	5.78	18.42
	P4	139.75	3.65	38.30	139.75	3.53	39.58	255.54	5.77	44.27	255.54	6.11	41.81	117.57	5.44	21.61	117.57	4.39	26.78	140.92	7.81	18.04	140.92	6.07	23.23
	P5	160.69	3.36	47.89	160.69	3.16	50.78	308.55	5.67	54.41	308.55	5.89	52.41	131.84	5.13	25.68	131.84	4.00	32.97	166.89	5.32	31.38	166.89	5.29	31.55
	P6	102.71	3.55	28.95	102.71	3.38	30.40	161.91	5.33	30.36	161.91	5.58	29.00	89.70	5.34	16.81	89.70	4.39	20.43	103.33	7.75	13.34	103.33	6.00	17.23
	P7	103.30	3.51	29.43	103.30	3.38	30.55	167.39	5.42	30.89	167.39	5.69	29.41	87.55	5.32	16.45	87.55	4.25	20.58	103.33	7.79	13.26	103.33	5.89	17.54
	P8	107.53	3.32	32.34	107.53	3.14	34.30	175.30	5.04	34.81	175.30	5.21	33.62	89.28	5.13	17.41	89.28	3.99	22.37	109.27	7.50	14.57	109.27	5.66	19.31
	P9	170.79	3.11	54.94	170.79	3.01	56.73	345.30	5.69	60.74	345.30	5.83	59.18	159.55	5.46	29.20	159.55	4.38	36.45	172.75	7.95	21.73	172.75	6.25	27.65
	P10	167.64	3.12	53.70	167.64	3.02	55.59	336.25	5.70	59.00	336.25	6.02	55.84	159.71	5.28	30.28	159.71	4.42	36.15	173.72	7.89	22.03	173.72	6.26	27.75
1500	P1	220.62	7.61	29.00	220.62	4.71	46.83	399.20	13.60	29.35	399.20	9.49	42.06	194.51	11.75	16.55	194.51	6.85	28.42	246.10	15.33	16.05	246.10	9.19	26.79
	P2	244.54	7.71	31.70	244.54	5.30	46.14	446.15	13.90	32.11	446.15	9.43	47.30	205.66	11.71	17.57	205.66	6.88	29.87	245.83	16.56	14.84	245.83	9.53	25.79
	P3	228.00	7.03	32.44	228.00	4.74	48.11	416.57	11.64	35.80	416.57	8.08	51.52	199.39	11.20	17.80	199.39	6.20	32.14	255.84	15.52	16.49	255.84	8.74	29.28
	P4	302.17	7.77	38.88	302.17	5.56	54.32	633.37	12.76	49.64	633.37	9.29	68.17	280.15	11.92	23.50	280.15	7.02	39.92	324.54	16.48	19.69	324.54	9.54	34.01
	P5	345.15	7.30	47.28	345.15	5.01	68.87	733.47	12.77	57.45	733.47	9.15	80.14	311.06	11.42	27.24	311.06	6.42	48.43	366.17	10.64	34.41	366.17	8.05	45.47
	P6	227.36	7.61	29.87	227.36	5.34	42.57	399.67	13.34	29.97	399.67	8.53	46.84	212.98	11.72	18.17	212.98	6.87	31.02	243.22	16.65	14.61	243.22	9.34	26.03
	P7	225.08	7.54	29.84	225.08	5.34	42.13	409.29	12.80	31.97	409.29	8.55	47.87	209.28	11.70	17.88	209.28	6.80	30.77	244.48	16.41	14.90	244.48	9.58	25.52
	P8	233.04	7.25	32.12	233.04	4.97	46.93	427.61	11.96	35.77	427.61	7.99	53.50	216.29	11.41	18.95	216.29	6.42	33.67	249.87	15.91	15.71	249.87	8.97	27.87
	P9	357.02	6.79	52.55	357.02	4.71	75.78	816.75	14.39	56.75	816.75	9.01	90.65	372.13	11.82	31.49	372.13	6.94	53.59	380.74	16.64	22.88	380.74	9.78	38.95
	P10	358.18	6.81	52.57	358.18	4.72	75.91	876.52	14.39	60.90	876.52	9.29	94.33	368.73	11.53	31.98	368.73	9.29	39.69	381.19	16.77	22.74	381.19	9.76	39.04
2000	P1	340.39	13.99	24.33	340.39	7.17	47.44	591.68	35.59	16.62	591.68	12.64	46.80	316.17	22.04	14.35	316.17	9.39	33.65	415.64	26.15	15.89	415.64	12.87	32.29
	P2	364.38	14.06	25.91	364.38	7.12	51.18	669.65	35.65	18.79	669.65	12.59	53.18	310.27	22.01	14.09	310.27	9.43	32.91	408.33	30.09	13.57	408.33	13.62	29.98
	P3	348.41	13.27	26.26	348.41	6.39	54.54	622.93	36.54	17.05	622.93	10.56	59.00	298.35	21.35	13.97	298.35	8.52	35.00	395.46	28.77	13.74	395.46	12.75	31.02
	P4	482.73	14.18	34.04	482.73	7.47	64.60	595.37	37.74	15.78	595.37	11.50	51.76	474.17	22.25	21.31	474.17	9.60	49.40	527.42	29.90	17.64	527.42	13.80	38.22
	P5	560.93	13.63	41.17	560.93	6.74	83.19	1187.25	38.48	30.85	1187.25	12.40	95.75	519.32	21.64	24.00	519.32	8.83	58.81	630.51	19.40	32.51	630.51	10.90	57.82
	P6	344.42	14.01	24.59	344.42	7.18	48.00	595.37	35.07	16.98	595.37	11.50	51.76	346.51	22.00	15.75	346.51	9.39	36.89	402.62	30.06	13.40	402.62	13.83	29.10
	P7	341.30	13.92	24.52	341.30	7.18	47.55	598.54	35.00	17.10	598.54	11.51	51.99	321.89	21.99	14.64	321.89	9.31	34.57	382.96	29.95	12.79	382.96	13.65	28.05
	P8	349.44	13.56	25.76	349.44	6.69	52.21	622.54	36.60	17.01	622.54	10.32	60.30	342.96	21.63	15.85	342.96	8.82	38.88	414.16	29.41	14.08	414.16	13.13	31.54
	P9	644.85	12.93	49.86	644.85	6.33	101.87	1322.22	35.67	37.07	1322.22	12.15	108.82	645.16	22.13	29.16	645.16	9.49	67.98	638.27	30.30	21.07	638.27	14.23	44.85
	P10	604.27	12.96	46.63	604.27	6.34	95.30	1322.06	35.84	36.89	1322.06	12.53	105.53	474.17	21.76	21.79	474.17	9.55	49.67	642.89	30.34	21.19	642.89	14.30	44.97

Advanced Computing (INCD/CNCA) through the FCT Advanced Computing Projects 2024.18424.CPCA.A1 and 2025.00279.CPCA.A2.

REFERENCES

[1] D.-C. Dang, T. Jansen, and P. K. Lehre, "Populations can be essential in tracking dynamic optima," *Algorithmica*, vol. 78, no. 2, pp. 660–680, 2017.

[2] R. Storn and K. Price, "Differential Evolution – a simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[3] N. Zhang, Y.-S. Chen, and J.-L. Wang, "Image parallel processing based on GPU," in *2010 2nd International Conference on Advanced Computer Control*, vol. 3. IEEE, 2010, pp. 367–370.

[4] K. V. Price, R. M. Storn, and J. A. Lampinen, *Differential Evolution: a practical approach to global optimization*, ser. Natural Computing Series. Berlin, Heidelberg: Springer, 2005.

[5] R. Tanabe and A. Fukunaga, "Success-history based parameter adaptation for Differential Evolution," in *2013 IEEE Congress on Evolutionary Computation*. IEEE, 2013, pp. 71–78.

[6] J. Zhang and A. C. Sanderson, "JADE: adaptive Differential Evolution with optional external archive," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 5, pp. 945–958, 2009.

[7] A. Viktorin, R. Senkerik, M. Pluhacek, T. Kadavy, and A. Zamuda, "Distance based parameter adaptation for Success-History based Differential Evolution," *Swarm and Evolutionary Computation*, vol. 50, p. 100462, 2019.

[8] J. Brest, S. Greiner, B. Boskovic, M. Mernik, and V. Zumer, "Self-adapting control parameters in Differential Evolution: a comparative study on numerical benchmark problems," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 6, pp. 646–657, 2006.

[9] Y. Wang, Z. Cai, and Q. Zhang, "Differential Evolution with composite trial vector generation strategies and control parameters," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 1, pp. 55–66, 2011.

[10] D. B. Kirk and W.-M. W. Hwu, *Programming massively parallel processors: a hands-on approach*, 2nd ed. Waltham, MA: Morgan Kaufmann, 2013.

[11] B. Silva, L. G. Lopes, and F. Mendonça, "Parallel GPU-acceleration of metaphorless optimization algorithms: application for solving large-scale nonlinear equation systems," *Applied Sciences*, vol. 14, no. 12, p. 5349, 2024.

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

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

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

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

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

[17] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Rev.*, vol. 59, no. 1, pp. 65–98, 2017.

[18] T. Besard, C. Foket, and B. De Sutter, "Effective extensible programming: Unleashing Julia on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 4, pp. 827–841, 2019.