Parallel Computing 126 (2025) 103160

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

GPU/CUDA-Accelerated gradient growth optimizer for efficient complex numerical global optimization

Qingke Zhang [a,*], Wenliang Chen [a], Shuzhao Pang [a], Sichen Tao [b], Conglin Li [a], Xin Yin [a]

[a] School of Information Science and Engineering, Shandong Normal University, Jinan 250358, China
[b] Faculty of Engineering, University of Toyama, Toyama-shi 930-8555, Japan

ARTICLE INFO

Keywords:
Parallel optimization
GPU acceleration
CUDA computing
Metaheuristic algorithm
Gradient-guided search
High-dimensional optimization

ABSTRACT

Efficiently solving high-dimensional and complex numerical optimization problems remains a critical challenge in high-performance computing. This paper presents the GPU/CUDA-Accelerated Gradient Growth Optimizer (GGO)—a novel parallel metaheuristic algorithm that combines gradient-guided local search with GPU-enabled large-scale parallelism. Building upon the Growth Optimizer (GO), GGO incorporates a dimension-wise gradient-guiding strategy based on central difference approximations, which improves solution precision without requiring differentiable objective functions. To address the computational bottlenecks of high-dimensional problems, a hybrid CUDA-based framework is developed, integrating both fine-grained and coarse-grained parallel strategies to fully exploit GPU resources and minimize memory access latency. Extensive experiments on the CEC2017 and CEC2022 benchmark suites demonstrate the superior performance of GGO in terms of both convergence accuracy and computational speed. Compared to 49 state-of-the-art optimization algorithms, GGO achieves top-ranked results in 67% of test cases and delivers up to 7.8× speedup over its CPU-based counterpart. Statistical analyses using the Wilcoxon signed-rank test further confirm its robustness across 28 out of 29 functions in high-dimensional scenarios. Additionally, in-depth analysis reveals that GGO maintains high scalability and performance even as the problem dimension and population size increase, providing a generalizable solution for high-dimensional global optimization that is well-suited for parallel computing applications in scientific and engineering domains.

1. Introduction

The efficient resolution of complex global optimization problems remains a foundational yet formidable challenge in modern scientific computing and engineering applications. Such problems are typically characterized by high-dimensional, non-convex, and multimodal search landscapes, often accompanied by intricate constraints and nonlinear objective interactions. These properties render traditional deterministic or gradient-based optimization techniques ineffective in achieving both high-quality solutions and computational scalability.

In recent decades, Evolutionary Computation (EC) and Swarm Intelligence (SI) algorithms have emerged as powerful population-based metaheuristic frameworks for tackling these challenges [1]. Drawing inspiration from natural evolution and collective behaviors in biological systems, EC and SI algorithms exhibit key properties such as self-organization, adaptability, and decentralized control. These characteristics enable them to navigate complex fitness landscapes by maintaining a dynamic balance between global exploration and local exploitation. Representative algorithms in this family include Genetic Algorithm (GA) [2], Particle Swarm Optimization (PSO) [3], Differential Evolution (DE) [4], Ant Colony Optimization (ACO) [5], and Artificial Bee Colony (ABC) [6], all of which have demonstrated successful applications across a wide range of domains, including intelligent control, resource scheduling, engineering design, and machine learning. Moreover, optimization has also been increasingly applied to practical domains such as industrial IoT intrusion detection and dynamic system design [7–9], further underscoring its broad applicability.

A critical factor influencing the performance of EC and SI algorithms is their ability to effectively manage the exploration–exploitation trade-off. Insufficient exploration may lead to premature convergence, while inadequate exploitation hinders convergence accuracy and efficiency. To address this, the Growth Optimizer (GO) [10] was recently introduced. Inspired by human cognitive development, GO simulates knowledge acquisition and self-reflection mechanisms to achieve a more structured and adaptive search process. Empirical results have shown that GO offers strong global search performance and broad applicability. However, a notable limitation lies in its lack of gradient

* Corresponding author.
E-mail address: tsingke@sdnu.edu.cn (Q. Zhang).

https://doi.org/10.1016/j.parco.2025.103160
Received 24 June 2025; Received in revised form 20 September 2025; Accepted 4 October 2025
Available online 10 October 2025

awareness, which can lead to suboptimal convergence rates and insufficient local refinement—particularly in complex or high-dimensional problem spaces where directional guidance is critical.

To overcome these deficiencies, this work proposes the Guided Growth Optimizer (GGO)—an enhanced version of GO that integrates a dimension-wise gradient-guiding mechanism based on central-difference numerical approximations. Unlike conventional gradient-based methods, the proposed approach does not rely on analytically available derivatives, making it suitable for non-differentiable, noisy, or black-box optimization scenarios. This gradient-guided factor enables more precise local updates without compromising the global search diversity, thereby significantly improving convergence speed, accuracy, and robustness. As a result, GGO effectively bridges the gap between metaheuristic exploration and gradient-informed exploitation, while preserving the scalability and flexibility inherent to population-based algorithms.

Beyond algorithmic enhancements, achieving computational scalability is critical when addressing high-dimensional and computationally expensive optimization tasks. Most population-based algorithms, including GO, exhibit a high degree of inherent parallelism, particularly through independent fitness evaluations and simultaneous population-level updates. These characteristics make such algorithms well-suited for acceleration on modern parallel hardware platforms—most notably, Graphics Processing Units (GPUs).

In this work, we develop a GPU-accelerated implementation of the proposed GGO algorithm using NVIDIA's Compute Unified Device Architecture (CUDA) programming model. To fully exploit both algorithmic and architectural parallelism, we design a hybrid parallel strategy that combines fine-grained and coarse-grained thread-level decomposition. This enables optimized memory access patterns, reduced communication overhead, and efficient workload distribution across thousands of CUDA threads, thereby facilitating scalable execution on large-scale optimization problems.

We validate the effectiveness of our method through extensive experiments on the CEC2017 and CEC2022 benchmark suites, covering diverse problem types and dimensionalities. The results demonstrate that GGO not only achieves superior solution accuracy and robustness but also delivers up to 7.8× speedup compared to its serial CPU-based counterpart, validated across 10 independent runs, ensuring statistical consistency (see Section 5.2). GGO consistently outperforms 49 state-of-the-art metaheuristic algorithms, and statistical analysis reveals significant advantages on 28 out of 29 high-dimensional test functions, highlighting its strength in both convergence quality and computational efficiency. In summary, the main contributions of this paper are as follows:

- We propose a gradient-guided extension of the Growth Optimizer that employs derivative-free directional approximations to enhance local search precision.
- We present a scalable, GPU-accelerated parallel implementation of the algorithm based on CUDA, incorporating a hybrid fine/coarse-grained strategy to maximize parallel throughput.
- We conduct comprehensive empirical evaluations on standardized benchmark functions, demonstrating that GGO achieves state-of-the-art performance in terms of accuracy, robustness, and runtime speedup.

The remainder of the paper is organized as follows. Section 2 reviews the fundamental concepts of GPU-based parallel computing. Section 3 introduces the baseline Growth Optimizer. Section 4 presents the proposed GGO algorithm and its parallel implementation. Section 5 provides experimental results and detailed analysis. Finally, Section 6 concludes the paper and outlines potential directions for future research.

2. GPU-based parallel computing

The Graphics Processing Unit (GPU) is a specialized processor designed for high-throughput parallel computing. With thousands of lightweight cores capable of executing concurrent threads, GPUs offer substantial performance advantages over traditional Central Processing Units (CPUs), particularly in data-intensive and compute-bound applications. NVIDIA's Compute Unified Device Architecture (CUDA) further enhances GPU utility by providing a scalable and programmable platform for general-purpose parallel processing. Together, GPU hardware and the CUDA programming model offer an effective framework for accelerating large-scale, population-based optimization algorithms.

In this study, we exploit GPU acceleration to parallelize the Gradient Growth Optimizer (GGO), thereby enabling it to tackle high-dimensional and computationally intensive optimization problems more efficiently. By leveraging the massive parallelism inherent in the GGO's population-level operations — such as fitness evaluation, solution updates, and gradient-guided perturbations — we construct a robust CUDA-based execution framework. This section provides a brief overview of the underlying GPU and CUDA technologies that support our parallel design. Implementation-specific optimizations will be detailed in Section 4.

2.1. Graphics Processing Unit (GPU)

Unlike CPUs, which are optimized for general-purpose sequential instruction execution, GPUs are architected to maximize throughput via massive parallelism. Originally developed to support graphics rendering, GPUs have evolved into essential components in high-performance computing (HPC), particularly in fields such as scientific simulation, computer vision, and machine learning.

Modern GPUs contain thousands of arithmetic logic units (ALUs) organized into Streaming Multiprocessors (SMs), enabling the simultaneous execution of a vast number of lightweight threads. Fig. 1 compares the architectural differences between CPU and GPU designs. While CPUs excel at executing complex control logic with large caches and fewer cores, GPUs achieve higher aggregate computational throughput through simplified control and deep thread-level parallelism.

A defining feature of GPU execution is the Single Instruction, Multiple Threads (SIMT) model, in which multiple threads execute the same instruction in parallel but on different data. This model, combined with efficient context switching and memory coalescing mechanisms, allows GPUs to maintain high occupancy and computational efficiency when properly utilized.

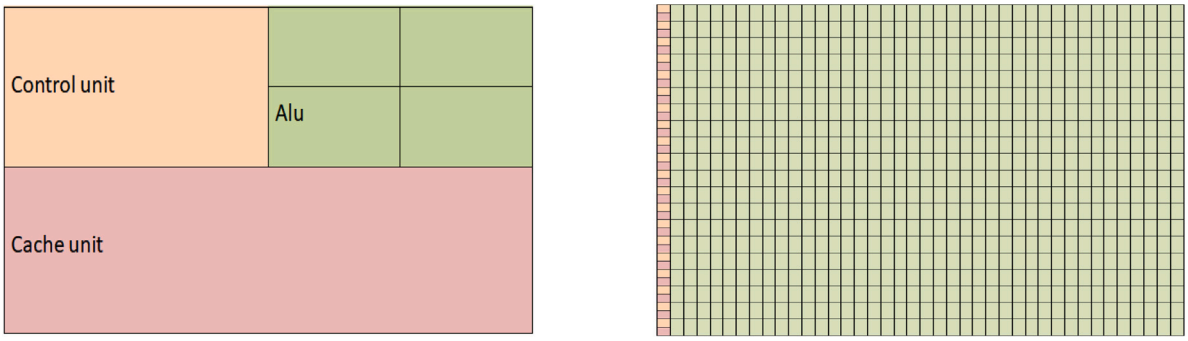
2.2. Compute Unified Device Architecture (CUDA)

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming interface developed by NVIDIA for general-purpose computing on GPUs. It enables developers to write highly parallel applications in familiar languages such as C, C++, FORTRAN, Java, and Python. In this work, we adopt C/C++ with CUDA as the implementation framework for GGO due to its maturity, flexibility, and the wide availability of optimized GPU libraries, including cuBLAS, cuFFT, and cuRAND.

CUDA programming is based on a hierarchical thread organization, where computations are structured as kernels launched over grids of blocks, each composed of multiple threads. This abstraction allows fine-grained control over data and task parallelism, which is essential for mapping the operations of GGO (e.g., particle-level and dimension-level updates) onto the GPU efficiently.

Efficient memory management is critical to CUDA performance. CUDA exposes a multi-tiered memory hierarchy (Fig. 2), including:

- Global memory: accessible by all threads but relatively slow; suitable for large data buffers.



(a) CPU (b) GPU

Fig. 1. Architectural comparison of CPUs and GPUs.

- Shared memory: fast, low-latency memory shared among threads within the same block; ideal for intra-block communication.
- Registers: the fastest storage, allocated per-thread; suitable for frequently accessed scalars or temporary results.

Performance optimization in CUDA relies heavily on a deep understanding and effective utilization of the GPU memory hierarchy and execution model. Specifically, minimizing access to global memory — which has high latency and limited bandwidth — is essential for reducing memory bottlenecks. Instead, maximizing the use of shared memory and registers, which offer much faster access speeds, can substantially improve data locality and throughput. Additionally, ensuring memory coalescing — where consecutive threads access consecutive memory locations — helps fully exploit memory bandwidth, while maximizing thread occupancy — the ratio of active threads per streaming multiprocessor — ensures that latency can be effectively hidden through warp scheduling.

Our parallel implementation of GGO is carefully designed with these optimization principles in mind. At the coarse-grained level, thread blocks are assigned to particles, allowing for parallel evaluation and update of candidate solutions. At the fine-grained level, dimension-wise operations such as gradient-guided perturbations are mapped to individual threads, enabling concurrent computations within each particle. Shared memory is used to store intermediate data that require intra-block synchronization, while registers handle frequently used variables to minimize instruction latency. Through these strategies, the GGO framework achieves high computational throughput and strong scalability across different problem dimensions, making it well-suited for large-scale and real-time optimization scenarios.

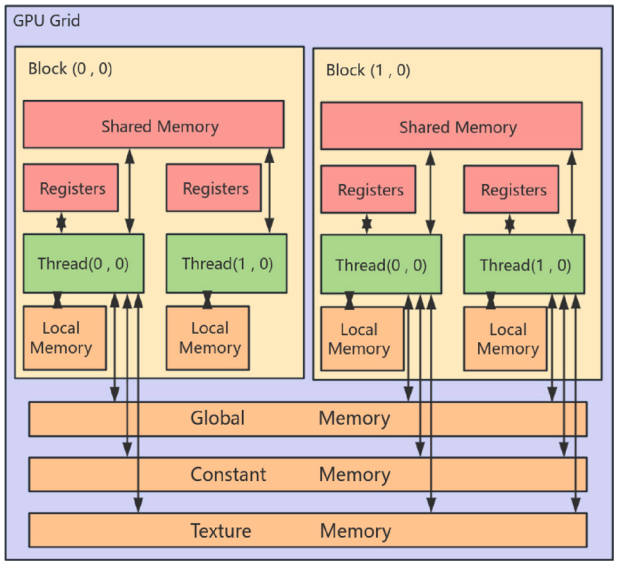


Fig. 2. CUDA/GPU memory hierarchy.

3. Growth Optimizer (GO)

The Growth Optimizer (GO) is a robust and adaptive metaheuristic algorithm inspired by the principles of human growth, including mechanisms of learning, self-reflection, and knowledge evolution. Designed to address both continuous and discrete optimization problems, GO excels in balancing global exploration and local exploitation, enabling it to navigate complex, multimodal, and high-dimensional search spaces with high effectiveness.

In GO, each solution candidate (or "individual") is mathematically encoded as a position vector $\vec{x}_i = (x_{i,1}, x_{i,2}, \ldots, x_{i,D})$, where $x_{i,D}$ denotes the value of the Dth dimension of the ith individual. The algorithm simulates growth by allowing individuals to gain knowledge from both elite and inferior solutions and to iteratively refine their own attributes through a structured learning–digestion–reflection cycle. This dual-source learning strategy improves convergence robustness and diversity preservation by integrating insights from both promising and underperforming regions in the search space. GO has been successfully applied to various real-world tasks, such as multi-threshold image segmentation (e.g., via Kapur's entropy method) and multiple sequence alignment based on Hidden Markov Models (HMMs). Its ability to maintain population diversity mitigates the risk of premature convergence and makes it suitable for solving complex global optimization problems.

3.1. Principles of the GO algorithm

As a novel metaheuristic approach, the GO framework consists of four stages: initialization, evaluation, learning, and reflection. The core process of the algorithm — namely, the updating and optimization of individuals — is realized by simulating human learning and self-reflection mechanisms. The overall workflow of these stages is illustrated in Fig. 3, and the detailed implementation of each stage will be further elaborated in the following sections.

Step 1: Initialization The algorithm begins by generating an initial population uniformly distributed in the defined search space. Each dimension of each individual is initialized using Eq. (1),

$$\mathbf{x}_{i,j} = lb + (ub - lb) \times \text{rand}(), \quad (i = 1, 2, \ldots, N,\ j = 1, 2, \ldots, D) \tag{1}$$

where the function rand() generates a uniformly distributed random number in $[0, 1]$, and lb and ub define the lower and upper bounds of the search space.

Step 2: Fitness Evaluation and Sorting Each individual is evaluated using a predefined fitness function $f(\cdot)$, providing a quantitative

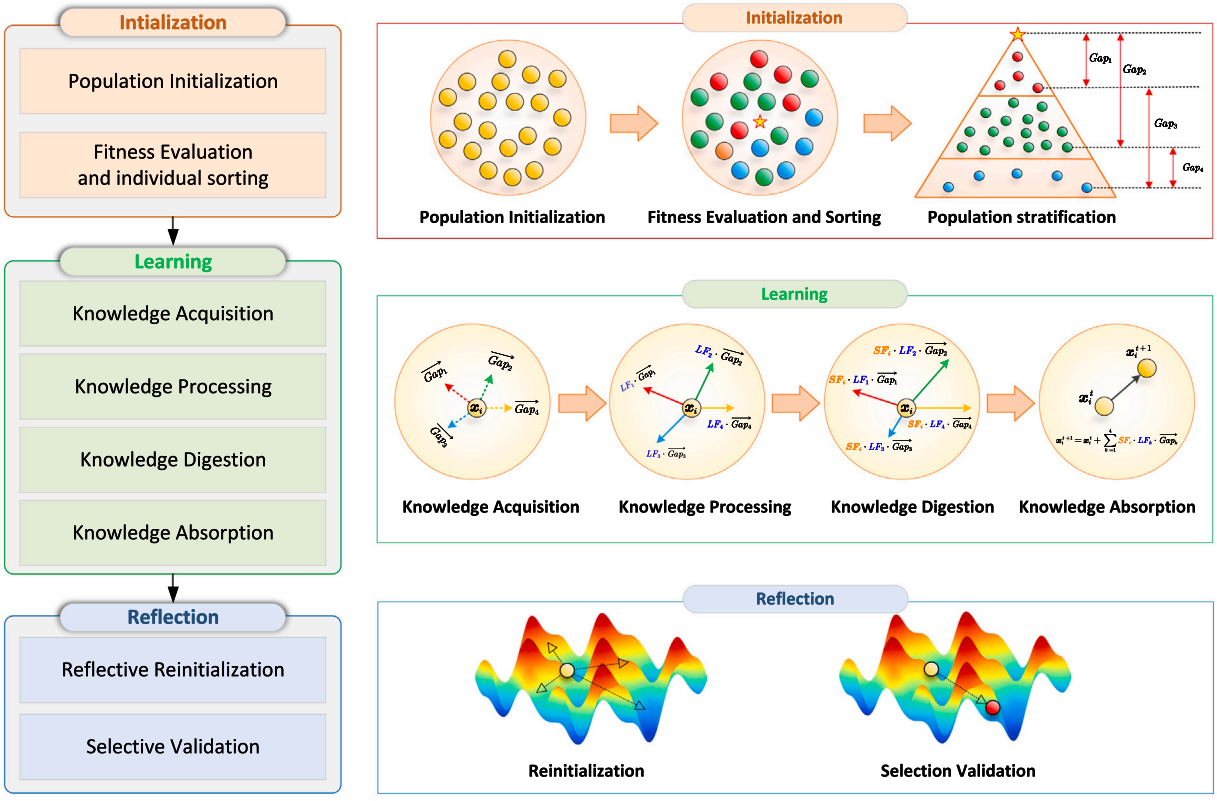


Fig. 3. Schematic flowchart of the main procedure in the GO algorithm.

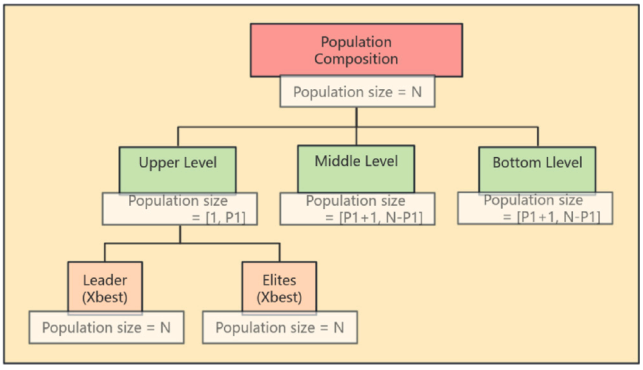


Fig. 4. The hierarchical distribution of individuals in the population.

measure of solution quality. The population is then sorted based on fitness values to facilitate elite selection and structured learning in subsequent phases.

Step 3: Learning Phase The learning phase forms the core of GO's adaptive behavior. Individuals are hierarchically categorized into three groups: elite leaders, average performers, and underperformers, as shown in Fig. 4. This stratification enables a diverse and informed exploration of the search space. The mathematical models for the learning phase is constructed through the following four steps :

(a) Knowledge Acquisition: To capture structural relationships within the population, GO defines four directional gap vectors that quantify differences among individuals: the gap between the global best individual and other elites; (ii) the gap between the best and the worst individuals; (iii) the gap between elite and poorly performing individuals; and (iv) the gap between two randomly selected individuals. These vectors, formally defined in Eq. (2), are denoted as Gap_k for $k = 1, 2, 3, 4$. Here $\mathbf{x}_{\text{best}}$ denotes the global best-performing individual, $\mathbf{x}_{\text{better}}$ denotes an elite peer, $\mathbf{x}_{\text{worse}}$ denotes a low-performing individual, and $\mathbf{x}_{L1}$ and $\mathbf{x}_{L2}$ denote two randomly selected individuals.

$$\begin{cases} \mathbf{Gap}_1 = \mathbf{x}_{\text{best}} - \mathbf{x}_{\text{better}} \\ \mathbf{Gap}_2 = \mathbf{x}_{\text{best}} - \mathbf{x}_{\text{worse}} \\ \mathbf{Gap}_3 = \mathbf{x}_{\text{better}} - \mathbf{x}_{\text{worse}} \\ \mathbf{Gap}_4 = \mathbf{x}_{L1} - \mathbf{x}_{L2} \end{cases} \tag{2}$$

The resulting gap vectors serve as informative search directions that help the algorithm assess population diversity and guide knowledge-driven updates.

(b) Knowledge Processing: Each gap vector contributes to the learning process through an associated learning factor LF_k, which determines the relative influence of each directional component on an individual's update. The learning factor for the kth gap is defined as in Eq. (3).

$$LF_k = \frac{\|\mathbf{Gap}_k\|}{\sum_{k=1}^{4} \|\mathbf{Gap}_k\|}, \quad (k = 1, 2, 3, 4) \tag{3}$$

This formulation normalizes the magnitude of each gap vector with respect to the total directional variation, assigning larger weights to more prominent structural differences. Consequently, directions associated with larger gaps exert a stronger influence on the individual's learning behavior, enabling adaptive, data-driven updates that reflect the underlying population distribution.

(c) Knowledge Digestion: The scaling factor SF_i quantifies the capacity of an individual i to assimilate acquired knowledge, serving as an adaptive weight in the learning process. It is computed as Eq. (4).

$$SF_i = \frac{GR_i}{GR_{\max}} \tag{4}$$

Here, GR_i denotes the growth rate of individual i, which reflects its responsiveness to past learning and adaptation efforts, while GR_{max}

is the maximum growth rate observed within the current population. A larger SF_i indicates a stronger demand for knowledge absorption, implying that the individual is in a phase of rapid learning or requires more intensive updates to improve its performance. This dynamic adjustment allows the algorithm to tailor the magnitude of updates based on each individual's learning potential.

(d) Knowledge Absorption: Based on the computed learning factor and scaling factor, the actual absorbed knowledge from each directional gap is calculated by Eq. (5).

$$\mathbf{KA}_k = SF_i \times LF_k \times \mathbf{Gap}_k, \quad k = 1, 2, 3, 4 \tag{5}$$

Each knowledge component $\mathbf{KA_k}$ represents a directional adjustment vector derived from the corresponding population relationship. The four components are then aggregated to update the individual's position by Eq. (6):

$$\mathbf{x}_i^{t+1} = \mathbf{x}_i^t + \mathbf{KA}_1 + \mathbf{KA}_2 + \mathbf{KA}_3 + \mathbf{KA}_4 \tag{6}$$

where t denotes the current iteration index. This update reflects the cumulative learning effect from diverse sources of knowledge within the population. To balance exploitation and diversity preservation, the newly generated solution x_i^{t+1} is accepted according to a selective retention strategy as shown in Eq. (7):

$$\mathbf{x}_i^{t+1} = \begin{cases} \mathbf{x}_i^{t+1} & \text{if } f(\mathbf{x}_i^{t+1}) < f(\mathbf{x}_i^t) \\ \begin{cases} \mathbf{x}_i^{t+1} & \text{if } r_1 < P_2 \\ \mathbf{x}_i^t & \text{else} \end{cases} & \text{else} \end{cases} \tag{7}$$

where P_2 is the retention probability, typically set to a small value (e.g., 0.001) to allow occasional acceptance of inferior solutions, thus enhancing the algorithm's global search capability and reducing the risk of premature convergence. This learning and update mechanism enables the GO algorithm to dynamically exploit structural knowledge from the population while retaining a controlled degree of randomness. As a result, it achieves a more accurate and robust local search process within a globally guided optimization framework.

Step 4: Reflection Phase: To further enhance population diversity and prevent stagnation in local optima, GO incorporates a probabilistic reflection mechanism that perturbs individual positions based on adaptive randomness and previously learned knowledge. This phase allows the algorithm to re-explore the solution space and validate the effectiveness of the absorbed information. The position update during reflection is defined by Eq. (8):

$$\mathbf{x}_i^{t+1} = \begin{cases} \begin{cases} lb + \mathbf{r}_4 \times (ub - lb) & \text{if } \mathbf{r}_3 < AF \\ \mathbf{x}_{i,j}^t + \mathbf{r}_5 \times (\mathbf{R}_j - \mathbf{x}_{i,j}^t) & \text{else} \end{cases} & \text{if } \mathbf{r}_2 < P_3 \\ \mathbf{x}_{i,j}^t & \text{else} \end{cases} \tag{8}$$

where $\mathbf{r}_2, \mathbf{r}_3, \mathbf{r}_4, \mathbf{r}_5$ denotes the rand vector, and each element in the vector is independent random numbers uniformly sampled from $[0, 1]$; lb and ub represent the lower and upper bounds of the search space; R_j is randomly selected reference value in dimension j; and P_3 is the reflection probability threshold, and is generally set to 0.3.

The adaptive reflection factor AF determines the likelihood of complete random reinitialization, and decays linearly with the number of function evaluations by Eq. (9):

$$AF = 0.01 + 0.99 \times \left(1 - \frac{FEs}{MaxFEs}\right) \tag{9}$$

where FEs denotes the current number of fitness evaluations and $MaxFEs$ is the maximum evaluation budget. As the algorithm advances, AF gradually decreases, reducing the frequency of disruptive random jumps and favoring more refined, knowledge-driven exploration.

After completing this reflection update, each individual additionally undergoes the selective retention strategy defined in Eq. (7), ensuring that only candidates exhibiting effective improvement are retained in the population.

This reflection strategy serves two complementary purposes: firstly, it introduces controlled exploration by allowing random re-initialization early in the search, and secondly, it supports knowledge-guided movement toward promising areas through differential updates. Together, these mechanisms improve the algorithm's ability to escape local optima and maintain a healthy balance between convergence and diversity throughout the optimization process.

The procedural framework of the GO algorithm is presented in **Algorithm 1**, where its initialization, learning, and reflection phases are systematically described in pseudocode.

Algorithm 1: Pseudocode of GO algorithm

Input: $N, D, lb, ub, MaxFEs, P_1, P_2, P_3$
Output: Global best solution **gbestx**
// Initialization
for $i \leftarrow 1$ **to** N **do**
  Generate population $\mathbf{x}_{i,j}$ by Eq. (1) ;
  Evaluate $f(\mathbf{x}_i)$ for individual i;
Set **gbestx** $\leftarrow \arg\min_i f(\mathbf{x}_i)$; $FEs \leftarrow N$
while $FEs \leq MaxFEs$ **do**
  // Role assignment by fitness
  Sort population; identify elites, average, underperformers; record $\mathbf{x}_{\text{best}}$;
  // Learning phase
  for $i \leftarrow 1$ **to** N **do**
    Update $\mathbf{x}_i$ by Eq. (2)–Eq. (6);
    Apply retention rule by Eq. (7);
    Update **gbestx** if improved; $FEs \leftarrow FEs + 1$;
  // Reflection phase
  Compute AF by Eq. (9);
  // (i) Reflective Reinitialization
  for $i \leftarrow 1$ **to** N **do**
    Reflect $\mathbf{x}_i$ by Eq. (8);
    Update **gbestx** if improved; $FEs \leftarrow FEs + 1$;
  // (ii) Selective Validation
  for $i \leftarrow 1$ **to** N **do**
    Verify $\mathbf{x}_i$ by applying selection rule Eq. (7);
    Update **gbestx** if improved; $FEs \leftarrow FEs + 1$;
return gbestx;

4. Gradient Growth Optimizer (GGO)

While the original Growth Optimizer (GO) algorithm demonstrates strong global search capabilities and a robust learning framework, its local exploitation efficiency remains limited due to the absence of directional guidance. This limitation often leads to slower convergence and reduced precision, particularly in complex or high-dimensional landscapes. To address this, we propose the Guided Growth Optimizer (GGO)—a novel extension of GO that integrates a gradient-guided search strategy and leverages GPU-based parallel computing to achieve scalable and efficient optimization.

4.1. Gradient-guided local search

In classical optimization, gradient descent is a foundational and widely adopted technique for local search. It iteratively updates the solution by following the negative gradient direction, which indicates the steepest descent of the objective function. When the objective function is smooth and differentiable, gradient descent can efficiently converge to a precise local or global optimum [11]. However, its applicability becomes severely limited in the presence of non-differentiable, discontinuous, or highly non-convex functions. In such scenarios, the lack of usable gradient information or the existence of deceptive gradient directions may cause premature convergence or search stagnation, as depicted in Fig. 5.

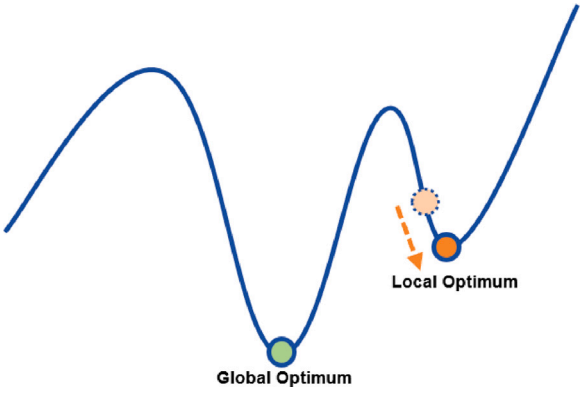


Fig. 5. Illustration of the limitations of traditional gradient descent in complex multimodal landscapes.

By contrast, metaheuristic algorithms — such as evolutionary and swarm intelligence methods — demonstrate strong robustness in solving such challenging problems. Their population-based, derivative-free, and stochastic nature enables them to explore non-smooth, non-convex, and black-box functions without requiring explicit gradient information. However, a common weakness of these methods lies in their limited local exploitation capability, often resulting in slower convergence and suboptimal accuracy in the vicinity of local minima.

To bridge this methodological gap, we propose a gradient-guided refinement strategy and incorporate it into the metaheuristic framework of the Growth Optimizer. The proposed mechanism enables dimension-wise directional adjustments using a central-difference approximation of partial derivatives. It introduces local sensitivity into the update process while preserving compatibility with non-differentiable, noisy, or complex fitness functions—thus achieving a hybrid balance between global exploration and local exploitation.

Extensive experiments on 41 benchmark functions from the CEC2017 and CEC2022 suites [12,13] demonstrate that this enhancement significantly improves both convergence speed and final solution quality. Furthermore, due to its fine-grained dimension-level structure, the gradient-guided mechanism is inherently suitable for parallel execution on GPUs, with directional derivative evaluations and updates naturally distributed across individual threads. This design reduces the theoretical time complexity from $O(n \times d)$ to near-constant time per thread, yielding excellent scalability in high-dimensional problems.

The core of this mechanism is defined by the following three equations:

1. **Directional Sensitivity Estimation**: Instead of requiring true gradients (unavailable in most black-box scenarios), GGO perturbs the solution slightly in both directions and computes the slope of the local landscape by Eq. (10).

$$\mathbf{gradient}\mathbf{x}_{i,j}^{t+1} = \frac{f(\mathbf{x}_{i,j,l}^{t}) - f(\mathbf{x}_{i,j,r}^{t})}{2 \times \epsilon} \tag{10}$$

   This equation estimates the partial directional derivative of the ith individual along dimension j using the central difference method. The two perturbed positions $\mathbf{x}_{i,j,l}^{t}$ and $\mathbf{x}_{i,j,r}^{t}$ represent slight deviations in opposite directions along dimension j. Compared to forward or backward difference, the central difference provides improved symmetry and higher estimation accuracy, particularly in noisy or rugged objective landscapes.

2. **Adaptive Gradient Scaling**: In order to ensure that early iterations emphasize broader movement (exploration), while later iterations focus on fine-tuning (exploitation), without the need for heuristic scheduling. GGO defines an individual-specific, time-adaptive step-size for gradient-guided movement by Eq. (11).

$$grad w_i^t = SF_i \times \alpha^t \tag{11}$$

   This formulation defines a decaying step-size coefficient for the gradient adjustment. The term SF_i is an individual-specific scaling factor (from the knowledge digestion phase), reflecting the individual's learning readiness. The decay term α^t ensures that the influence of the gradient diminishes over time—supporting exploration in early iterations and refined local exploitation in later stages.

3. **Dimension-Wise Position Update**: Each individual is updated independently based on the computed gradient direction and the adaptive step size by Eq. (12).

$$\mathbf{x}_{i,j}^{t+1} = \mathbf{x}_{i,j}^{t} + grad w_i^t \times \mathbf{gradient}\mathbf{x}_{i,j}^{t} \tag{12}$$

   This update rule adjusts each decision variable independently, using the direction and strength of the local gradient estimate. Since each dimension is updated separately, the algorithm gains fine-grained control over the search process, improving its ability to adapt in non-separable, high-dimensional, or anisotropic landscapes.

In summary, Eqs. (10)–(12) constitute a mathematically simple yet functionally powerful gradient-aware local search operator that seamlessly integrates into the GGO framework. This component introduces directional exploitation capabilities into the traditionally gradient-free metaheuristic architecture, thereby addressing one of the most persistent limitations in swarm intelligence and evolutionary algorithms. Specifically, the proposed mechanism: Enhances local convergence without sacrificing generality;Requires no analytical gradient, maintaining compatibility with black-box functions;Supports parallel execution at the dimension-thread level in CUDA, enabling efficient deployment on GPUs;Enables GGO to inherit the benefits of both metaheuristic and gradient-based methods in a mathematically consistent and computationally scalable framework.

4.2. Parallelization strategies in GGO algorithm

To fully exploit the computational capabilities of heterogeneous platforms, the GGO algorithm is designed with a CPU–GPU collaborative architecture, where the CPU is responsible for task scheduling and overall control, while the GPU handles large-scale parallel computation. As pointed out in [14], data transfer and task communication between host (CPU) and device (GPU) are often the primary performance bottlenecks. Therefore, minimizing communication overhead, constructing an efficient heterogeneous computing model, and ensuring optimized coordination between CPU and GPU are essential for achieving high parallel performance. Fig. 6 presents the overall workflow of the parallel GGO algorithm. In this architecture, the CPU initializes global parameters and controls the iteration loop, while all major algorithmic components — including population initialization, fitness evaluation, gradient-guided updates, and position reflection — are executed entirely on the GPU. This design ensures that data computation and exchange are conducted within the GPU memory hierarchy, significantly reducing latency and maximizing runtime efficiency. As the CPU's role is minimal and infrequent, the remainder of this section focuses on the CUDA-based parallelization of the GGO algorithm.

In CUDA programming, a large number of threads are launched simultaneously to perform computations in parallel, with each thread assigned to a specific subset of data based on its unique thread index. The mapping of threads to data is determined by their position within blocks and grids, as defined by the following equations, Eqs. (13) and (14):

$$\texttt{idx} = \texttt{threadIdx.x} + \texttt{blockDim.x} \times \texttt{blockIdx.x} \tag{13}$$

$$\texttt{position} = \texttt{d\_positions[idx]} \tag{14}$$

Here, the `threadIdx.x` indicates a thread's index within a block, `blockIdx.x` refers to the block's index within the grid, and `blockDim.x`

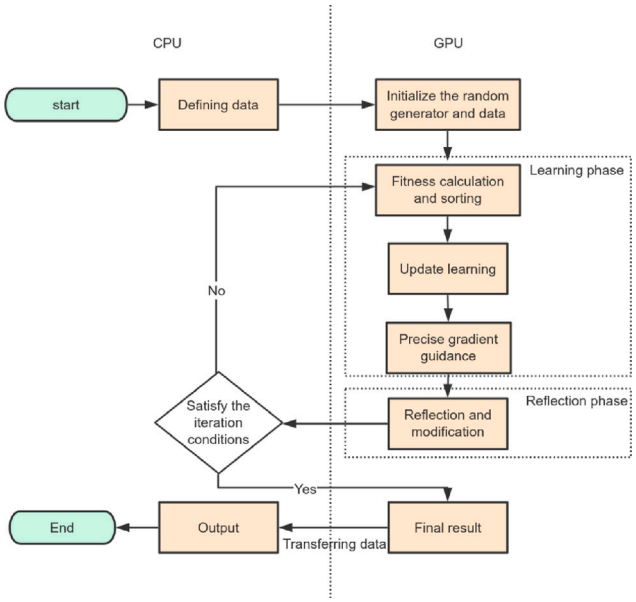


Fig. 6. The main workflow of the parallel GGO algorithm.

defines the number of threads per block. These values collectively determine the global thread index `idx`, which is then used to access specific data such as particle positions from global memory (`d_positions`). Thread-local variables such as `position` are typically stored in fast-access registers to reduce memory latency.

Although this thread mapping mechanism simplifies parallel programming, increasing the number of threads does not always yield better performance. For example, in fitness evaluation, the most efficient configuration is achieved when each thread processes exactly one particle. Over-allocation of threads may result in inefficient memory use and reduced occupancy. Table 1 lists the memory layout of key variables used in the parallel implementation.

To maximize resource utilization and adapt to different computational loads, GGO employs a hybrid of coarse-grained and fine-grained parallelization strategies, as illustrated in Fig. 7. In the coarse-grained mode, each particle is assigned to a single thread for full-dimensional updates, while in the fine-grained mode, each dimension of a particle is assigned to a separate thread, allowing for highly parallelized dimension-wise operations. Theoretically, the fine-grained scheme enables a maximum speedup of $n \times m$ for n particles and m dimensions, assuming ideal hardware conditions and full thread occupancy.

The step-by-step GPU implementation of GGO is provided in **Algorithm 2**, which details the allocation of parallel resources for each computational stage, including sorting, learning, gradient-guided refinement, and reflection.

Step 1: Initialization To initialize the particle population, GGO leverages NVIDIA's `cuRAND` library, which offers efficient parallel random number generation. A pseudo-random number generator is selected due to its speed and simplicity, with each thread assigned a unique seed to ensure randomness across particles and dimensions. The initialization is implemented as a dedicated CUDA kernel that sets the random states and generates initial positions in parallel. The position of each particle is initialized by Eq. (15),

$$\begin{aligned} \mathbf{x}_{\text{idx}} &= \text{curand(idx)} \times (ub - lb) + lb, \\ &\text{idx} = 1, 2, \ldots, N, \end{aligned} \tag{15}$$

Here, all particles are stored in a flattened $1D$ array, with the thread index defined as `idx = i × D + j`, representing the jth dimension of the ith particle.

Algorithm 2: Pseudocode of GGO Algorithm

Input: $N, D, lb, ub, MaxFEs, P_2, P_3$
Output: Global best solution **gbestx**

```
// Initialization
Initialize x_{i,j} in [lb, ub]; evaluate f(x_i); set
  gbestx ← arg min_i f(x_i); FEs ← N
while FEs ≤ MaxFEs do
    // Particle sorting & indexing
    Sort population by f(·); build role/index arrays
    // Learning phase
    for i ← 1 to N  in parallel do
        Update x_i via knowledge
          acquisition/processing/digestion/absorption using Eq.
          (2)–Eq. (6); apply retention Eq. (7);
        update gbestx if improved; FEs ← FEs + 1
    // Gradient-guided refinement
    for (i, d)  in parallel do
        Estimate direction by Eq. (10);
        scale by Eq. (11);
        update x_{i,d} by Eq. (12);
        FEs ← FEs + 1
    // Reflection phase
    Compute AF by Eq. (9);
    for i ← 1 to N  in parallel do
        Reflect x_i by Eq. (8);
        update gbestx if improved; FEs ← FEs + 1
    // Fitness refresh
    Recompute f(x_i) for all i;
    update gbestx if improved
return gbestx
```

Step 2: Fitness Evaluation and Sorting Fitness evaluation is a key component for guiding the optimization process. In GGO, fitness computation is performed entirely on the GPU using a coarse-grained parallel strategy, where each thread evaluates the fitness of one particle. The evaluation is expressed by Eq. (16).

$$\mathbf{fitness}_{\text{idc}} = \text{getFit}(\mathbf{X}_{\text{idc}}) \tag{16}$$

To sort the population based on fitness, a parallel *merge sort* algorithm is employed, achieving a time complexity of $O(n \cdot logn)$. This allows for efficient ranking and group assignment within the learning phase.

Step 3: Learning Phase In this phase, particle positions are updated based on the hierarchical learning mechanism of GO. The GGO implementation adopts a fine-grained parallel strategy, where each particle-dimension pair is updated independently. Shared memory is used to store intermediate operators and fitness values within blocks to reduce memory access latency. The block size is aligned with the number of dimensions D, ensuring full coverage and high memory throughput during updates.

Step 4: Gradient-Guided Refinement The gradient guidance mechanism operates at the dimension level, making it naturally suitable for fine-grained parallelism. Each thread computes a central difference-based directional derivative for a single dimension of a particle, and stores intermediate values (e.g., partial gradients and guidance factors) in registers to maximize computational speed. The independence of per-dimension operations ensures excellent parallel scalability across high-dimensional spaces.

Step 5: Reflection Phase The reflection mechanism stochastically perturbs or resets particle positions to maintain diversity and prevent premature convergence. This process is fully parallelized using the `cuRAND` library and follows a fine-grained model where each thread

Table 1
Parallel thread settings for each parameter.

Variable name	Shape	Variable name	Shape
Position: x	N × D	Fitness value: fit	N × 1
Temporary position: newx, gradx	N × D	Temporary fitness value: newfit, gradfit	N × 1
Gradient: gradient	N × D	Class gap: Gap	4 × D
Parameters: SF, gradw, rate, randid	N × 1		

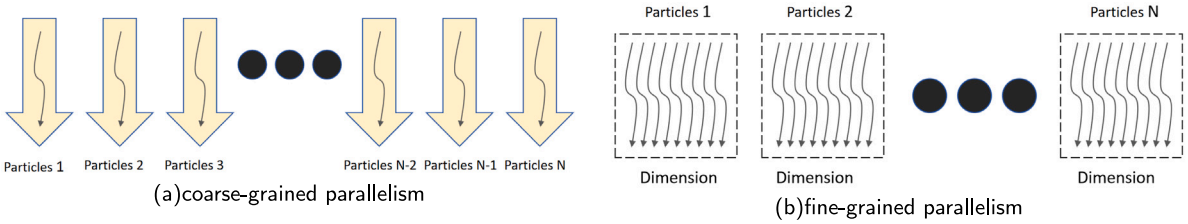


Fig. 7. The data parallel mode includes (a) coarse-grained parallelism and (b) fine-grained parallelism.

updates a specific dimension. Thread-level randomization enables asynchronous reflection, further improving the exploration behavior of the algorithm.

By combining coarse-grained strategies for population-level tasks (e.g., sorting, fitness evaluation) and fine-grained strategies for dimension-level updates (e.g., learning, gradient refinement, reflection), GGO achieves highly efficient parallelism. This hybrid design significantly enhances the real-time performance and scalability of metaheuristic optimization, particularly in large-scale and high-dimensional applications.

5. Experiments and analysis of results

5.1. Optimization performance evaluation

5.1.1. Experimental setup and baseline algorithms

To rigorously evaluate the optimization performance of the proposed GGO algorithm, we conduct comprehensive experiments on two widely used benchmark suites: CEC2017 and CEC2022. These benchmarks include a diverse range of test functions — covering unimodal, multimodal, hybrid, and composition categories — which are designed to reflect various real-world problem characteristics and are commonly adopted in advanced metaheuristic performance evaluations [12,13]. Specifically, the test sets comprise 41 benchmark functions from CEC2017 at dimensions 30, 50, and 100, as well as the full CEC2022 suite at dimensions 10 and 20. This setup enables a thorough assessment of scalability and robustness across different levels of problem complexity and dimensionality.

To validate the effectiveness of GGO, we benchmark it against nearly 49 cutting-edge metaheuristic algorithms, including well-established methods such as the Marine Predators Algorithm (MPA) [15], Supply–Demand-Based Optimization (SDO) [16], and the original Growth Optimizer (GO) [10]. All baseline algorithms are implemented using their standard configurations as recommended in the original publications. The maximum number of function evaluations (MaxFEs) is uniformly adopted as the stopping criterion to ensure fairness. Each algorithm is independently run 10 times per benchmark function to account for the stochastic nature of metaheuristics, and the mean and standard deviation of the obtained fitness values are recorded for performance comparison.

To determine the statistical significance of performance differences, we apply two widely recognized non-parametric tests: the Friedman rank-sum test and the Wilcoxon signed-rank test; both were conducted at a 5% significance level. These statistical tools provide a robust basis for validating whether observed improvements by GGO are consistent and not due to random variation.

All serial implementations of the GGO algorithm were developed in MATLAB R2024b and executed on a computing platform equipped with an Intel Core i7-12700 CPU @ 2.10 GHz, 16 GB RAM, and a 64-bit Windows operating system. The detailed parameter settings for all compared algorithms are summarized in Table 2.

5.1.2. Performance on CEC2017 and CEC2022 benchmark functions

Due to the large number of functions included in the CEC2017 benchmark suite, we report detailed results only for the top 10 performing algorithms in high-dimensional settings to maintain clarity and conciseness. Table 3 presents the comparative results of these top 10 algorithms (out of 50 tested) on the 100-dimensional CEC2017 test set. Likewise, Tables 4 and 5 report performance results for the CEC2022 suite under 10- and 20-dimensional configurations, respectively.

For each test function, the reported metrics include the mean and standard deviation (`mean` and `std`) of the fitness values over 10 independent runs. To assess the relative performance of GGO against each comparison algorithm, a symbolic indicator $h \in \{+, =, -\}$ is used, where the symbol + indicates that GGO significantly outperforms the compared algorithm; the symbol = denotes statistically equivalent performance; and the symbol − signifies inferior performance, respectively. Furthermore, a cumulative summary of the comparative performance is provided using the $w/t/l$ notation, where w denotes the number of functions that GGO achieves significantly better results; t indicates the number of functions with no statistically significant difference; and l denotes the number of functions that GGO performs worse.

The numerical experimental results demonstrate that in high-dimensional settings ($D = 100$), GGO consistently outperforms the original GO and other state-of-the-art algorithms in terms of mean fitness values. Specifically, GGO achieves superior performance on 10 out of 29 test functions in the CEC2017 suite and shows notable advantages on 3 functions in the CEC2022 suite. According to the h-value analysis, GGO yields slightly inferior results on certain functions (e.g., $F5$, $F7$, $F21$, $F22$, $F24$, and $F26$) compared to individual algorithms such as AEFA. Nevertheless, GGO maintains robust performance across the majority of functions, particularly excelling on $F1$, $F3$, $F12$, $F25$, and $F29$.

A similar trend is observed in the CEC2022 benchmark. In the 10-dimensional setting, performance drops are observed on $F7$ and $F10$, while in the 20-dimensional setting, GGO underperforms on $F6$ and $F10$. These weaknesses are largely attributed to the intrinsic difficulty of certain *multimodal functions*, such as $F7$ and $F10$, which are characterized by a large number of local optima with intricate distributions. Although the gradient-guided mechanism in GGO is effective at accelerating convergence, it may reduce global exploration capacity in landscapes with shallow or closely spaced local minima. In such cases,

Table 2
Overview of comparative algorithms and corresponding parameter settings.

NO	Algorithms	Parameters settings
1	Genetic Algorithm (GA) [2]	pc=0.8,pm=02
2	Simulated Annealing (SA) [17]	T_0=0.1, α=0.99,c=5, μ=0.5, σ=0.1 × (ub-1b)
3	Cultural Algorithm (CA) [18]	pAccept=0.35, α=0.3
4	Particle Swarm Optimization (PSO) [3]	w=1, w_α=0.99, c_1=1.5, c_2=2.0
5	Differential Evolution (DE) [4]	F=0.5,CR=0.9
6	Harmony Search (HS) [19]	HMCR=0.9,PAR=0.1,FW=0.02 × (ub-1b), FW_{damp}=0.995
7	Ant Colony Optimization (ACO) [5]	q=0.5, ζ=1
8	Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [20]	σ= 0.3
9	Shuffled Frog-Leaping Algorithm (SFLA) [21]	Memeplexes=5,Offsprings=3,Step=2,Run=5
10	Artificial Bee Colony (ABC) [6]	a=1,L=round(0.6 × N × D)
11	Biogeography-Based Optimization (BBO) [22]	KeepRate=0.2, α =0.9,Mutation=0.1, σ=0.02 × (ub-1b)
12	Firefly Algorithm (FA) [23]	α=0.2, β=2, γ=1, δ=0.05 × (ub-lb),Damping Ratio=0.98
13	Cuckoo Search (CS) [24]	P_a=0.25, α=1, β=1.5, σ_v=1
14	Gravitational Search Algorithm (GSA) [25]	G_0=100, α= 20,Rpower=1
15	Bacterial Foraging optimization (BFO) [26]	c=25,s=4,r=4,p=0.5,Sr=N/2
16	Brain Storm Optimization (BSO) [27]	m=5, P_{5a}=0.2, P_{6b}=0.8, P_{6bi}=0.4, P_{6c}=0.5,k=20, μ=0, σ=l
17	Teaching–Learning-Based Optimization (TLBO) [28]	T_F=l or 2
18	Differential Search (DS) [29]	P_1 =0.3 × rand, P_2=0.3 × rand
19	Flower Pollination Algorithm (FPA) [30]	p=0.8
20	Ali Baba And The Forty Thieves Algorithm (AFT) [31]	a=ceil((nnoThieve-1).*rand(noThieves,1))
21	Grey Wolf Optimizer (GWO) [32]	α=2-2 × (FEs/MaxFEs)
22	Moth–Flame Optimization (MFO) [33]	r=l-(FEs/MaxFEs),t=r+(l-r) × rand,b=1
23	Lightning Search Algorithm (LSA) [34]	maxchtime=10,direct=sign(unifrnd(−1,1))
24	Stochastic Fractal Search (SFS) [35]	MDN=2,Walk=l
25	Whale Optimization Algorithm (WOA) [36]	b=1, a_1=2-(2 × FEs/MaxFEs), a_2=-l-(FEs/MaxFEs)
26	Jaya [37]	No parameters
27	Sine Casine Algorithm (SCA) [38]	a=2
28	Multi-Verse Optimizer (MVO) [39]	WEP_{max}=l, WEP_{min}=0.2
29	Salp Swarm Algorithm (SSA) [40]	$c_1 = 2 \times exp((4 \times FEs/MaxFEs)^2)$
30	Coyote Optimization ANgorithm (COA) [41]	N_c=5, N_p=8
31	Atom Search Optimization (ASO) [42]	α=50, β=0.2
32	Artificial Electric Field Algorithm (AEFA) [43]	K_0=500, α=30
33	Harris Hawks Optimization (HHO) [44]	E_0= 2 × rand-1, E_1=2-2 × (FEs/MaxFEs)
34	Artificial Ecosystem-based Optimization (AEO) [45]	a = rand × (FEs/MaxFEs),c = randn/2 × |randn|
35	Supply–Demand-Based Optimization (SDO) [16]	a_1 = 2×((MaxFEs-FEs+ 1)/MaxFEs)
36	Manta Ray Foraging Optimization (MRFO) [46]	S=2, r_1= rand,Coef=FEs/MaxFEs
37	Gaining Sharing Knowledge (GSK) [47]	p=0.1, k_r=0.9, k_f=0.5
38	Equilibrium Optimizer (EO) [48]	V=1, a_1=2, a_2=1,GP=0.5
39	Coronavirus Herd Immunity Optimizer (CHIO) [49]	MaxAge=100,CO =1,BRr=0.05
40	Heap-Based Optimizer (HBO) [50]	$cycles$ = ⌊MaxIt/25⌋,degree=3
41	Marine Predators Algorithm (MPA) [15]	FADs=0.2,P=0.5
42	Arithmetic Optimization Algorithm (AOA) [51]	MOP_{Max}= 1, MOP_{Min}=0.2, α=5, μ=0.499
43	Runge kutta optimizer (RUN) [52]	a=20,b=12
44	weighted mean of vector (INFO) [30]	I=rand([2.5])
45	White Shark Optimizer (WSO) [53]	f=[0.07,0.75], τ=4.11,p=[0.5,1.5], a_0= 6.25, a_1=100, a_2=0.0005
46	Honey Badger Algorithm (HBA) [54]	β=6,c=2,flag=-1 or 1
47	Dwarf Mongoose Optimization Algorithm (DMOA) [55]	b=3,Lp=0.6,peep= =2
48	Snake Optimizer (SO) [56]	T_1=0.25, T_2=0.6, C_1= 0.5, C_2=0.05, C_3=2
49	Growth Optimizer (GO) [10]	P_1=5, P_2=0.001, P_3=0.3
50	Gradient Growth Optimizer (GGO) [57]	$Enum$=10, $Bnum$=4 P_1=5, P_2=0.001, P_3=0.3

the algorithm tends to converge rapidly to suboptimal regions, limiting its ability to continue exploring for the global optimum.

Despite these few limitations, GGO demonstrates **significant overall robustness and efficiency** across both benchmark suites. The gradient-guiding factor enhances the algorithm's ability to maintain population diversity while improving the precision of local exploitation. Compared to the original GO, the proposed GGO achieves more reliable convergence, better avoids premature stagnation, and exhibits improved performance in both solution quality and search efficiency.

5.1.3. Convergence analysis

Convergence curves provide a visual representation of an algorithm's convergence speed and its ability to avoid local optima. This subsection selects eight representative functions from the CEC 2017 and 2022 test suites for analysis, namely the unimodal $F1$, simple multimodal $F4$, hybrid function $F20$, and composite function $F25$ from CEC2017, and the unimodal $F1$, simple multimodal $F4$, hybrid function $F6$, and composite function $F12$ from CEC2022. The unimodal function $F1$ is utilized to assess the convergence rate and precision of algorithms during the global search process. The simple multimodal function $F4$ examines the algorithm's ability to explore when confronted with multiple local optima. The hybrid functions $F20$ and $F6$, which integrate various function characteristics, pose a higher demand on the algorithm's comprehensive performance. The composite functions $F25$ and $F12$ further increase the complexity of the problem, effectively evaluating the robustness and efficiency of algorithms in solving highly complex optimization problems. Conducting experimental analysis on these representative functions can provide valuable references for subsequent algorithm improvement and optimization. Fig. 8 presents the test curves for these eight functions across five dimensions, specifically D = 10, D = 20, D = 30, D = 50, and D = 100. Among them, the 10-dimensional and 20-dimensional problems correspond to the CEC2022 test set, while the remaining dimensions correspond to the CEC2017 test set. Given the large number of algorithms involved in the tests, the convergence curves in Fig. 8 only include the statistical data of the top ten algorithms.

The convergence curves indicate that, compared with GO and other algorithms, GGO generally exhibits superior convergence accuracy and efficiency This suggests that the gradient-guiding factor is well-suited to the original GO algorithm, enabling rapid convergence to potential

Table 3
Results of CEC2017 benchmark test functions of GGO with top ten ranked algorithms on 100-Dimension.

Function	Item	AEFA	ASO	COA	DE	EO	GO	GSK	MPA	SFS	GGO
F1	mean	1.85E+03	2.88E+03	5.24E+03	1.09E+04	3.11E+03	3.32E+03	6.21E+03	1.85E+04	6.49E+03	**0.00E+00**
	std	1.27E+03	3.07E+03	4.39E+03	1.61E+04	3.88E+03	5.73E+03	5.00E+03	1.19E+04	9.51E+03	**0.00E+00**
	h	+	+	+	+	+	+	+	+	+	=
F3	mean	3.69E+05	2.52E+04	4.50E+05	1.40E+05	6.57E+04	1.59E+01	1.67E+04	9.27E+02	4.02E+03	**2.61E−13**
	std	1.48E+04	6.61E+03	4.96E+04	2.22E+04	1.01E+04	1.43E+01	1.21E+04	3.83E+02	2.15E+03	**4.28E−13**
	h	+	+	+	+	+	+	+	+	+	=
F4	mean	2.30E+02	3.78E+02	2.46E+02	2.60E+02	1.75E+02	2.14E+02	1.74E+02	3.04E+02	2.20E+02	**4.11E+01**
	std	3.00E+01	7.42E+01	4.57E+01	3.33E+01	4.59E+01	4.93E+01	2.87E+01	9.27E+00	4.33E+01	8.74E+01
	h	+	+	+	+	=	=	=	+	+	=
F5	mean	1.50E+02	2.55E+02	4.68E+02	2.13E+02	4.99E+02	1.57E+02	1.94E+02	5.46E+02	6.97E+02	1.98E+02
	std	1.23E+01	8.74E+00	8.64E+01	5.87E+01	9.92E+01	5.36E+00	2.34E+01	8.56E+01	9.54E+01	2.60E+01
	h	−	+	+	=	+	−	=	+	+	=
F6	mean	9.32E−02	4.13E−01	1.68E−01	3.32E+00	2.33E+01	6.64E−02	3.53E+00	1.14E+01	1.47E+01	**4.25E−02**
	std	9.41E−02	3.88E−01	2.97E−02	1.59E+00	9.83E+00	1.12E−02	1.36E+00	3.27E+00	6.43E+00	2.59E−02
	h	=	+	+	+	+	=	+	+	+	=
F7	mean	2.10E+02	2.69E+02	7.32E+02	4.61E+02	9.09E+02	2.97E+02	3.95E+02	1.21E+03	1.38E+03	3.16E+02
	std	6.25E+00	1.24E+01	1.58E+02	2.87E+01	1.48E+02	3.53E+01	7.17E+01	1.29E+02	2.95E+02	2.91E+01
	h	−	−	+	+	+	=	=	+	+	=
F8	mean	1.72E+02	2.96E+02	4.99E+02	2.05E+02	5.05E+02	1.57E+02	2.02E+02	5.44E+02	6.89E+02	2.04E+02
	std	2.23E+01	2.99E+01	8.63E+01	2.40E+01	3.42E+01	2.71E+01	4.31E+01	2.61E+01	1.46E+02	3.11E+01
	h	=	+	+	=	+	=	=	+	+	=
F9	mean	3.62E+01	3.08E−01	8.60E+03	8.75E+02	1.16E+04	4.12E+02	1.44E+03	1.36E+04	1.64E+04	1.16E+00
	std	6.47E+01	2.84E−01	1.91E+03	3.67E+02	2.34E+03	1.72E+02	1.56E+03	5.06E+03	2.34E+03	7.13E−01
	h	=	=	+	+	+	+	+	+	+	=
F10	mean	7.45E+03	1.57E+04	1.29E+04	2.98E+04	1.48E+04	8.43E+03	2.90E+04	1.29E+04	1.23E+04	8.79E+03
	std	1.11E+03	6.19E+03	9.36E+02	6.02E+02	9.35E+02	7.59E+02	7.23E+02	8.74E+02	2.03E+03	1.26E+03
	h	=	+	+	+	+	=	+	+	+	=
F11	mean	1.25E+04	6.26E+02	3.64E+02	2.43E+02	7.06E+02	2.22E+02	2.93E+02	8.91E+02	3.73E+02	2.56E+02
	std	5.71E+03	9.63E+01	8.66E+01	5.66E+01	2.25E+02	5.47E+01	9.23E+01	1.87E+02	9.62E+01	8.08E+01
	h	+	+	=	=	+	=	=	+	=	=
F12	mean	7.41E+06	3.96E+05	2.30E+06	7.52E+05	1.51E+06	2.00E+05	3.59E+05	2.98E+07	5.68E+05	**8.75E+04**
	std	5.93E+06	1.08E+05	3.90E+05	4.21E+05	4.99E+05	9.02E+04	1.29E+05	1.42E+07	1.51E+05	**4.23E+04**
	h	+	+	+	+	+	+	+	+	+	=
F13	mean	2.98E+04	7.40E+03	4.09E+03	6.47E+03	1.29E+04	4.08E+03	3.34E+03	2.53E+04	3.06E+03	**4.16E+02**
	std	5.06E+03	5.01E+03	3.31E+03	7.96E+03	5.70E+03	2.41E+03	1.39E+03	1.22E+04	4.41E+03	**3.45E+02**
	h	+	+	+	+	+	+	+	+	=	=
F14	mean	5.13E+05	7.01E+04	2.02E+04	3.03E+04	2.48E+05	5.19E+02	1.46E+03	3.43E+03	7.15E+03	**2.59E+02**
	std	1.67E+05	1.00E+05	2.06E+04	1.26E+04	1.06E+05	7.56E+02	7.22E+02	5.22E+03	8.04E+03	**1.33E+02**
	h	+	+	+	+	+	=	+	+	+	=
F15	mean	1.25E+03	1.45E+03	2.20E+03	3.21E+03	2.93E+03	1.23E+03	2.64E+03	6.74E+03	1.95E+03	**3.86E+02**
	std	7.93E+02	9.77E+02	2.32E+03	2.76E+03	2.23E+03	7.92E+02	3.83E+03	4.75E+03	1.23E+03	**2.72E+02**
	h	=	+	=	+	+	=	=	+	+	=
F16	mean	3.05E+03	3.00E+03	4.11E+03	2.43E+03	3.74E+03	1.71E+03	1.46E+03	2.95E+03	3.55E+03	1.48E+03
	std	3.59E+02	3.58E+02	7.38E+02	9.21E+02	9.05E+02	2.44E+02	2.31E+02	8.95E+02	4.84E+02	2.42E+02
	h	+	+	+	=	+	=	=	+	+	=
F17	mean	2.49E+03	1.97E+03	2.54E+03	2.68E+03	3.48E+03	1.94E+03	2.23E+03	2.12E+03	3.04E+03	**1.12E+03**
	std	6.06E+02	3.23E+02	3.20E+02	1.32E+03	8.03E+02	8.01E+02	1.48E+03	4.24E+02	8.58E+02	**2.83E+02**
	h	+	+	+	=	+	+	+	+	+	=
F18	mean	5.32E+05	1.13E+05	1.02E+06	1.11E+05	5.62E+05	5.48E+03	4.38E+04	6.20E+04	2.01E+05	7.79E+03
	std	3.38E+05	3.97E+04	1.23E+06	6.06E+04	9.75E+04	3.54E+03	1.91E+04	2.88E+04	1.38E+05	1.03E+04
	h	+	+	+	+	+	=	+	+	+	=
F19	mean	1.24E+03	1.61E+03	1.71E+03	5.64E+03	7.10E+03	7.24E+03	1.79E+03	6.47E+03	6.20E+03	**5.89E+02**
	std	8.87E+02	1.85E+03	1.77E+03	9.71E+03	5.87E+03	4.76E+03	2.85E+03	3.64E+03	5.55E+03	**8.60E+02**
	h	=	=	=	=	+	+	=	+	+	=
F20	mean	2.21E+03	3.19E+03	2.90E+03	3.17E+03	3.02E+03	1.70E+03	3.66E+03	2.16E+03	2.19E+03	**1.01E+03**
	std	2.59E+02	9.08E+02	2.54E+02	5.73E+02	6.08E+02	9.79E+02	1.07E+03	1.42E+02	3.90E+02	**1.15E+02**
	h	+	+	+	+	+	=	+	+	+	=
F21	mean	3.82E+02	5.25E+02	7.34E+02	4.47E+02	6.34E+02	3.53E+02	4.06E+02	5.53E+02	7.18E+02	4.19E+02
	std	1.82E+01	5.39E+01	9.88E+01	4.67E+01	4.83E+01	2.11E+01	3.84E+01	2.94E+01	7.54E+01	2.52E+01
	h	=	+	+	=	+	−	=	+	+	=
F22	mean	1.00E+02	1.30E+04	1.48E+04	3.03E+04	1.58E+04	8.15E+03	2.98E+04	1.10E+04	1.44E+04	1.13E+04
	std	7.09E−13	1.50E+03	1.69E+03	6.59E+02	1.60E+03	7.06E+02	1.29E+03	6.12E+03	2.10E+03	1.14E+03
	h	−	=	+	+	+	−	+	=	+	=
F23	mean	7.58E+02	1.71E+03	7.87E+02	7.89E+02	9.33E+02	6.35E+02	7.28E+02	7.89E+02	1.08E+03	6.76E+02
	std	3.65E+01	2.71E+02	4.03E+01	5.12E+01	3.03E+01	2.53E+01	2.95E+01	2.68E+01	7.10E+01	2.95E+01
	h	+	+	+	+	+	=	+	+	+	=
F24	mean	9.71E+02	3.60E+03	1.39E+03	1.19E+03	1.28E+03	1.08E+03	1.12E+03	1.40E+03	1.70E+03	1.13E+03
	std	1.24E+01	1.26E+03	8.02E+01	4.95E+01	1.19E+02	2.38E+01	6.79E+01	5.47E+01	1.13E+02	3.83E+01
	h	−	+	+	=	+	−	=	+	+	=
F25	mean	7.91E+02	1.00E+03	8.27E+02	7.98E+02	7.82E+02	8.13E+02	8.44E+02	8.01E+02	7.59E+02	**6.66E+02**
	std	5.62E+01	6.24E+01	9.00E+01	7.89E+01	4.14E+01	7.31E+01	1.86E+01	7.24E+01	4.69E+01	4.17E+01
	h	+	+	+	+	+	+	+	+	+	=

(*continued on next page*)

Table 3 (*continued*).

F26	mean	1.09E+03	6.14E+03	8.49E+03	7.03E+03	6.72E+03	5.40E+03	6.24E+03	8.85E+03	1.64E+04	5.49E+03
	std	1.77E+03	3.32E+03	7.45E+02	6.76E+02	3.63E+03	4.29E+02	5.52E+02	1.28E+03	7.18E+03	**2.06E+02**
	h	–	=	+	+	=	=	+	+	=	=
F27	mean	6.38E+02	2.12E+03	8.03E+02	7.91E+02	8.05E+02	7.45E+02	8.41E+02	8.56E+02	1.11E+03	6.57E+02
	std	1.85E+01	1.39E+03	6.41E+01	4.82E+01	6.78E+01	5.10E+01	3.13E+01	4.88E+01	2.46E+02	1.90E+01
	h	=	+	+	+	+	+	+	+	+	=
F28	mean	5.71E+02	7.36E+02	6.23E+02	5.88E+02	5.59E+02	5.39E+02	5.75E+02	6.21E+02	5.95E+02	**5.03E+02**
	std	3.55E+01	8.83E+01	5.19E+01	2.88E+01	5.09E+01	2.50E+01	2.13E+01	1.68E+01	3.56E+01	3.97E+01
	h	+	+	+	+	=	=	+	+	+	=
F29	mean	2.99E+03	2.55E+03	3.37E+03	2.25E+03	3.37E+03	1.76E+03	2.01E+03	3.14E+03	3.44E+03	**1.17E+03**
	std	4.20E+02	3.81E+02	4.22E+02	2.73E+02	5.10E+02	2.22E+02	3.19E+02	3.79E+02	2.93E+02	2.94E+02
	h	+	+	+	+	+	+	+	+	+	=
F30	mean	1.01E+05	1.30E+04	7.51E+03	7.21E+03	2.22E+04	2.95E+03	6.33E+03	1.44E+05	8.43E+03	**2.40E+03**
	std	3.81E+04	4.94E+03	5.01E+03	1.29E+03	6.80E+03	9.89E+02	5.83E+03	3.59E+04	4.75E+03	**1.17E+02**
	h	+	+	+	+	+	=	+	+	+	=
	w/t/l	16/8/5	24/4/1	26/3/0	21/8/0	26/3/0	10/15/4	19/10/0	28/1/0	26/3/0	0/29/0
	Friedman Rank	12.21	15.4	14.78	13.71	16.46	6.06	11.45	16.18	15.79	2.92

Table 4
Results of CEC2022 benchmark test functions of GGO with top ten ranked algorithms on 10-Dimension.

Function	Item	ASO	COA	CS	GO	GSK	HBO	MPA	SDO	SFS	GGO
F1	mean	3.00E+02	3.00E+02	3.00E+02	3.00E+02	3.00E+02	3.00E+02	3.00E+02	3.00E+02	3.00E+02	**3.00E+02**
	std	0.00E+00	4.27E−09	4.19E−05	0.00E+00	0.00E+00	0.00E+00	1.16E−12	0.00E+00	9.13E−12	**0.00E+00**
	h	=	+	+	=	=	=	+	=	+	=
F2	mean	4.12E+02	4.05E+02	4.00E+02	4.03E+02	4.05E+02	4.07E+02	4.00E+02	4.02E+02	4.00E+02	**4.00E+02**
	std	2.62E+01	3.79E+00	1.17E−01	3.94E+00	4.88E+00	1.57E+00	7.51E−10	2.18E+00	7.27E−08	**0.00E+00**
	h	+	+	+	=	=	+	+	+	+	=
F3	mean	6.00E+02	6.00E+02	6.01E+02	6.00E+02	6.00E+02	6.00E+02	6.00E+02	6.00E+02	6.00E+02	**6.00E+02**
	std	1.59E−07	1.50E−04	2.29E−01	6.89E−06	8.04E−14	0.00E+00	1.90E−04	1.97E−04	1.14E−13	5.68E−14
	h	+	+	+	=	=	=	+	=	+	=
F4	mean	8.05E+02	8.18E+02	8.15E+02	8.06E+02	8.17E+02	8.16E+02	8.07E+02	8.11E+02	8.13E+02	**8.03E+02**
	std	3.19E+00	1.15E+01	4.24E+00	1.57E+00	6.20E+00	7.97E+00	1.51E+00	6.32E+00	4.16E+00	1.78E+00
	h	=	+	+	=	+	+	+	+	+	=
F5	mean	9.00E+02	9.01E+02	9.00E+02	9.00E+02	9.00E+02	9.00E+02	9.00E+02	9.00E+02	9.00E+02	**9.00E+02**
	std	0.00E+00	1.72E+00	2.92E−01	0.00E+00	0.00E+00	1.14E−13	1.92E−09	0.00E+00	0.00E+00	**0.00E+00**
	h	=	=	+	=	=	+	+	=	=	=
F6	mean	2.11E+03	1.80E+03	1.81E+03	1.80E+03	1.80E+03	2.92E+03	1.80E+03	1.80E+03	1.80E+03	**1.80E+03**
	std	1.93E+02	7.25E−01	1.34E+00	7.03E−02	6.24E−01	8.29E+02	9.17E−02	8.56E−01	1.23E+00	1.33E−01
	h	+	=	+	+	=	+	=	=	=	=
F7	mean	2.02E+03	2.00E+03	2.01E+03	2.01E+03	2.01E+03	2.00E+03	2.01E+03	2.01E+03	2.00E+03	**2.00E+03**
	std	9.31E+00	1.15E−12	3.11E+00	9.53E+00	1.11E+01	1.91E−05	9.80E+00	1.03E+01	4.91E−02	5.41E−01
	h	+	–	+	=	=	–	+	+	=	=
F8	mean	2.22E+03	2.21E+03	2.21E+03	2.20E+03	2.20E+03	2.20E+03	2.20E+03	2.22E+03	2.20E+03	2.21E+03
	std	9.68E+00	1.09E+01	4.32E+00	3.69E−01	9.57E+00	4.31E+00	2.28E+00	8.86E+00	4.84E−01	8.75E+00
	h	+	=	=	=	=	=	=	=	=	=
F9	mean	2.53E+03	2.53E+03	2.48E+03	2.53E+03	2.53E+03	2.53E+03	2.53E+03	2.53E+03	2.53E+03	2.53E+03
	std	1.46E+00	0.00E+00	6.34E+01	0.00E+00	0.00E+00	0.00E+00	0.00E+00	2.27E−13	0.00E+00	**0.00E+00**
	h	+	=	=	=	=	=	=	=	=	=
F10	mean	2.50E+03	2.48E+03	2.50E+03	2.50E+03	2.50E+03	2.40E+03	2.50E+03	2.50E+03	2.50E+03	2.52E+03
	std	6.42E−02	4.48E+01	3.50E−02	3.36E−02	2.12E−02	5.23E+00	3.27E−02	5.21E−02	8.13E−02	4.69E+01
	h	=	=	=	=	=	–	=	=	=	=
F11	mean	2.60E+03	2.60E+03	2.60E+03	2.72E+03	2.60E+03	2.66E+03	2.60E+03	2.60E+03	2.60E+03	**2.60E+03**
	std	2.27E−13	6.37E−11	5.87E−05	1.26E+02	2.27E−13	8.24E+01	5.11E−06	0.00E+00	3.94E−13	2.27E−13
	h	+	+	+	=	=	=	+	=	=	=
F12	mean	2.87E+03	2.86E+03	2.86E+03	2.86E+03	2.86E+03	2.86E+03	2.86E+03	2.86E+03	2.86E+03	**2.86E+03**
	std	3.36E+00	1.59E+00	1.60E+00	1.60E+00	1.51E+00	1.09E+00	3.36E−01	1.06E+00	8.51E−01	1.53E+00
	h	+	+	=	=	+	=	=	+	+	=
	w/t/l	8/4/0	6/5/1	8/4/0	1/11/0	2/10/0	4/6/2	7/5/0	4/8/0	5/7/0	0/12/0
	Friedman Rank	19.82	15.92	19.39	10.53	11.67	14.41	12.67	13.3	11.97	6.9

optimal values through gradient guidance while maintaining robust global search capabilities. However, it is observed from Figure *m* that GSK demonstrates relatively better convergence performance on $F20$. This may be attributed to the fact that the knowledge-sharing mechanism of the GSK algorithm is particularly effective for handling the hybrid function $F20$, which features multiple local optima and complex modality distributions. These characteristics align with the challenges posed by the different function properties within hybrid functions, thereby yielding favorable convergence results on this specific function.

5.1.4. Statistical analysis of results

Boxplots are a widely used tool for visualizing data distributions and are particularly robust to outliers, making them well-suited for assessing the stability and dispersion of algorithmic performance. Fig. 9 presents boxplots for eight representative functions at various dimensions (as previously described), summarizing the results from 10 independent runs for each algorithm to provide a comprehensive view of performance variability.

The height of each box reflects the spread of fitness values: a shorter box indicates less fluctuation, implying greater algorithmic stability. The central line denotes the median fitness, while the whiskers extend

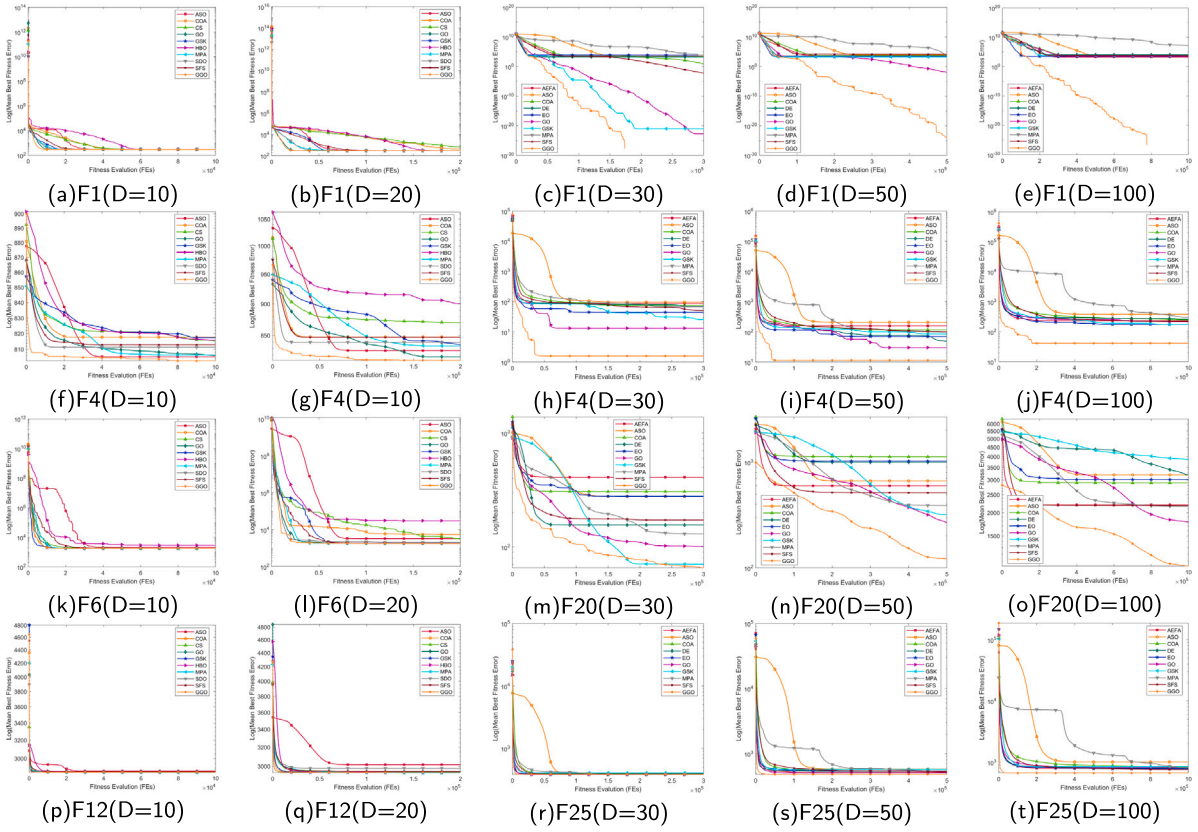


Fig. 8. The convergence curves of GGO with top ten ranked algorithms on four groups of problems.

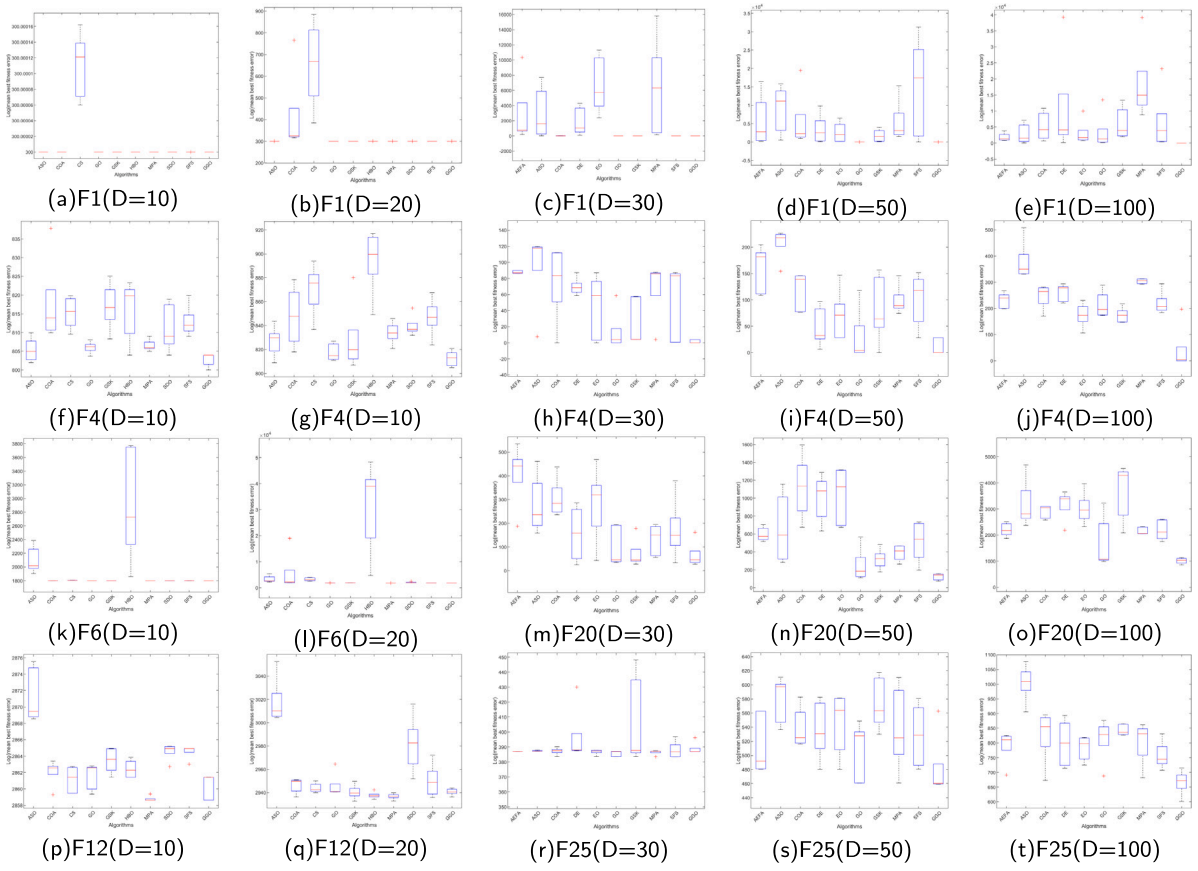


Fig. 9. The convergence curves of GGO with top ten ranked algorithms on four groups of problems.

Table 5
Results of CEC2022 benchmark test functions of GGO with top ten ranked algorithms on 20-Dimension.

Function	Item	ASO	EO	GO	GSK	HBO	MPA	SDO	SFS	TLBO	GGO
F1	mean	3.00E+02	3.00E+02	3.00E+02	3.00E+02	3.00E+02	3.00E+02	3.00E+02	3.00E+02	3.00E+02	**3.00E+02**
	std	2.84E−14	2.54E−04	4.92E−14	4.92E−14	2.93E−06	7.56E−05	2.84E−14	4.45E−06	7.52E−14	**2.84E−14**
	h	=	+	=	+	+	+	=	+	+	=
F2	mean	4.51E+02	4.53E+02	4.31E+02	4.20E+02	4.47E+02	4.38E+02	4.20E+02	4.26E+02	4.43E+02	4.38E+02
	std	2.90E+01	8.25E+00	2.47E+01	2.61E+01	2.23E+00	2.10E+01	2.69E+01	2.23E+01	2.64E+01	2.16E+01
	h	=	+	=	=	=	=	=	=	=	=
F3	mean	6.00E+02	6.00E+02	6.00E+02	6.00E+02	6.00E+02	6.00E+02	6.02E+02	6.00E+02	6.04E+02	**6.00E+02**
	std	1.55E−06	3.32E−04	2.80E−04	4.50E−06	0.00E+00	1.85E−02	2.03E+00	5.68E−14	4.29E+00	1.47E−05
	h	=	+	+	=	=	+	+	=	+	=
F4	mean	8.27E+02	8.32E+02	8.18E+02	8.29E+02	8.95E+02	8.34E+02	8.40E+02	8.47E+02	8.32E+02	**8.12E+02**
	std	1.27E+01	1.39E+01	7.22E+00	2.94E+01	2.71E+01	9.08E+00	8.77E+00	1.57E+01	7.41E+00	**6.51E+00**
	h	=	+	=	=	+	+	+	+	+	=
F5	mean	9.00E+02	9.03E+02	9.00E+02	9.00E+02	9.00E+02	9.04E+02	9.27E+02	9.80E+02	9.18E+02	**9.00E+02**
	std	0.00E+00	3.28E+00	2.62E−01	3.11E−13	1.74E−01	2.82E+00	2.27E+01	1.48E+02	8.39E+00	**0.00E+00**
	h	=	+	+	+	+	+	+	=	+	=
F6	mean	3.31E+03	4.26E+03	1.82E+03	1.86E+03	3.10E+04	1.80E+03	2.06E+03	1.82E+03	5.29E+03	1.81E+03
	std	1.29E+03	4.35E+03	2.18E+01	2.66E+01	1.71E+04	1.96E+00	2.21E+02	1.25E+01	5.15E+03	5.65E+00
	h	+	+	=	+	+	−	+	=	+	=
F7	mean	2.04E+03	2.05E+03	2.02E+03	2.02E+03	2.03E+03	2.03E+03	2.04E+03	2.02E+03	2.05E+03	**2.02E+03**
	std	6.36E+00	1.89E+01	8.42E+00	3.00E−01	8.67E+00	5.27E+00	1.66E+01	1.27E+01	1.23E+01	4.45E+00
	h	+	+	=	=	=	+	+	=	+	=
F8	mean	2.22E+03	2.22E+03	2.22E+03	2.23E+03	2.23E+03	2.22E+03	2.22E+03	2.22E+03	2.23E+03	**2.22E+03**
	std	7.92E−01	2.17E−01	8.48E−01	7.44E−01	3.82E−01	3.11E−01	1.50E+00	4.23E−01	3.23E+00	6.82E−01
	h	+	+	+	+	+	=	+	=	=	=
F9	mean	2.49E+03	2.48E+03	2.48E+03	2.48E+03	2.48E+03	2.48E+03	2.48E+03	2.48E+03	2.48E+03	**2.48E+03**
	std	1.46E+00	6.82E−13	2.27E−13	3.22E−13	0.00E+00	2.11E−08	3.78E−12	6.82E−12	2.10E−12	**0.00E+00**
	h	+	+	=	+	=	+	=	+	+	=
F10	mean	2.50E+03	2.90E+03	2.55E+03	2.53E+03	2.41E+03	2.50E+03	2.53E+03	2.50E+03	2.70E+03	2.52E+03
	std	4.36E−02	6.05E+02	7.25E+01	5.63E+01	8.91E+00	4.65E−02	7.04E+01	7.18E−02	4.48E+02	5.45E+01
	h	=	=	=	=	−	=	=	=	=	=
F11	mean	2.98E+03	2.92E+03	2.92E+03	2.94E+03	2.90E+03	2.90E+03	2.90E+03	2.92E+03	2.86E+03	**2.78E+03**
	std	4.47E+01	4.47E+01	4.47E+01	5.48E+01	6.02E−13	1.73E+02	0.00E+00	4.47E+01	1.53E+02	1.64E+02
	h	+	+	=	+	=	=	=	=	=	=
F12	mean	3.02E+03	2.95E+03	2.95E+03	2.94E+03	2.94E+03	2.94E+03	2.98E+03	2.95E+03	3.00E+03	**2.94E+03**
	std	1.99E+01	1.19E+01	1.06E+01	6.18E+00	2.89E+00	2.66E+00	2.37E+01	1.43E+01	5.43E+01	2.87E+00
	h	+	=	=	=	=	=	+	=	+	=
	w/t/l	6/6/0	10/2/0	3/9/0	6/6/0	5/6/1	6/5/1	7/5/0	3/9/0	8/4/0	0/12/0
	Friedman Rank	19.53	18.21	11.93	11.86	16.32	14.13	16.48	12.83	21.49	6.79

to the minimum and maximum observed values, excluding outliers, which are plotted as individual points. As illustrated in Fig. 9, performance variability tends to increase with dimensionality across all algorithms. GGO exhibits the most stable behavior, particularly at $D = 30$ and $D = 50$. At $D = 100$, its variance becomes comparable to that of other algorithms.

The last two plots in the fourth row of Fig. 9 correspond to composite function $F12$ from the CEC2022 suite. At a lower dimensionality ($D = 10$), the Marine Predators Algorithm (MPA) exhibits superior stability, likely due to the smaller search space and reduced distance between local and global optima, which favors MPA's exploitation strategy. In such settings, MPA can effectively leverage its biologically inspired search behavior to quickly locate the global optimum. However, as dimensionality increases, the complexity and multimodality of the composite functions rise significantly, reducing MPA's effectiveness. In contrast, GGO maintains robust performance by combining an adaptive learning strategy with the proposed gradient-guided refinement mechanism, which enables broad search during early iterations and fine-tuned convergence in later stages. This hybrid capability leads to greater robustness and stability, especially in high-dimensional problem settings.

To statistically validate these observations, we employed non-parametric tests including the Friedman test and the Wilcoxon signed-rank test to assess differences in algorithmic performance. Table 6 reports the results of the Friedman test, which is particularly suitable for comparing multiple algorithms across repeated trials without assuming normality or homoscedasticity—assumptions typically required by parametric tests such as repeated-measures ANOVA.

As shown in the table, GGO achieves the highest average rank (4.762) across all benchmark functions, outperforming its predecessor GO and other competing algorithms. In contrast, algorithms such as GSK, MPA, and SFS experience performance degradation with increasing dimensionality, indicating that they are better suited to low-dimensional problems. Conversely, algorithms like SDO, COA, and EO improve as dimensionality increases, suggesting stronger adaptation to large-scale scenarios. Notably, GGO consistently ranks among the top performers across all dimensions, and its performance advantage becomes more pronounced in higher dimensions, underscoring its scalability and robustness in tackling complex, high-dimensional optimization problems.

The Wilcoxon signed-rank test is a widely used non-parametric statistical method for comparing two related samples or paired observations. As a distribution-free alternative to the paired t-test, it is particularly appropriate when the assumption of normality is violated. Given the large number of algorithms included in our study, we report results only for the top six performers to maintain clarity. Table 7 presents pairwise comparisons of GGO against GO, GSK, MPA, SFS, and SDO across different problem dimensionalities ($D = 10$, 20, 30, 50, and 100).

The results reveal that GGO consistently achieves statistically superior performance over the baseline methods, with its advantage becoming more prominent as dimensionality increases. In lower-dimensional settings ($D = 10$ to 30), GGO performs comparably to GO, with only marginal improvements. However, in higher-dimensional settings — particularly at $D = 100$ — GGO exhibits substantial gains, outperforming GO in 10 out of 14 test functions.

In addition to GO, GGO also demonstrates clear superiority over the other advanced algorithms. At $D = 100$, GGO outperforms GSK in 19 cases, MPA in 28 cases, and SFS in 26 cases. These results underscore the effectiveness of the gradient-guided local refinement mechanism

Table 6
The Friedman test ranks for GGO and the compared algorithms.

id	Algorithms	D=10	D=20	D=30	D=50	D=100	Mean rank
1	GGO	6.9	6.79	3.93	3.27	2.92	4.762
2	GO	10.53	11.93	6.51	5.6	6.06	8.126
3	GSK	11.67	11.86	8.34	10.21	11.45	10.706
4	MPA	12.67	14.13	10.77	11.05	16.18	12.96
5	SFS	11.97	12.83	11.63	15.22	15.79	13.488
6	SDO	13.3	16.48	17.21	19.05	17.23	16.654
7	COA	15.92	18.63	17.57	18.22	14.78	17.024
8	ASO	19.82	19.53	15.28	15.74	15.4	17.154
9	DE	23.86	23.72	12.32	14.69	13.71	17.66
10	EO	21.09	18.21	18.89	18.72	16.46	18.674
11	AEFA	27.46	22.33	19.3	15.43	12.21	19.346
12	HBO	14.41	16.32	22.71	23.06	23.17	19.934
13	DS	21.95	20.52	17.2	19.57	20.92	20.032
14	FA	25.07	20.32	20.23	18.23	17.28	20.226
15	SO	21.73	24.98	21.86	20.37	18.51	21.49
16	TLBO	20.89	21.49	22.14	23	23.86	22.276
17	CS	19.39	22.82	20.63	23.1	25.48	22.284
18	SA	25.47	20.53	20.84	21.63	23.86	22.466
19	AFT	18.73	20.3	26.7	24.69	24.11	22.906
20	INFO	23.58	24.31	22.63	24.58	21.68	23.356
21	PSO	23.07	24.23	24.72	23.11	22.37	23.5
22	HBAO	26.11	27.24	23.59	23.77	17.59	23.66
23	AEO	23.85	25.79	24.81	24.13	21.26	23.968
24	FPA	24.45	26.23	21.7	24.33	23.61	24.064
25	MRFO	23.79	24.73	25.47	25.84	21.01	24.168
26	SFLA	20.94	18.57	19.04	18.3	45.22	24.414
27	WSO	20.05	24.62	25.72	26.99	26.73	24.822
28	CHIO	28	24.87	24.94	22.81	23.97	24.918
29	MVO	27.58	27.95	25.79	23.49	20.01	24.964
30	HS	25.15	21.45	22.41	25.54	30.96	25.102
31	BBO	26.58	26.33	25.64	23.76	23.36	25.134
32	GA	31.53	27.63	29.38	25.74	21.68	27.192
33	SSA	29.4	27.88	29.48	25.86	23.54	27.232
34	ABC	21.28	23.18	29.09	30.1	33.63	27.456
35	DMOA	22.57	22.31	29.37	30.19	33.08	27.504
36	RUN	32.25	29.7	26.33	27.49	24.97	28.148
37	ACO	22.65	22.93	29.67	31.09	36.21	28.51
38	GSA	32.49	31.91	31.24	28.65	23.59	29.576
39	CMA-ES	35.28	31.67	27.68	27.2	28.34	30.034
40	LSA	29.47	27.48	32.7	31.98	29.15	30.156
41	GWO	30.98	32.75	30.91	30.02	31.55	31.242
42	BSO	30.28	30.39	36.94	35.72	32.2	33.106
43	CA	33.88	37.61	37.79	39.04	42.46	38.156
44	MFO	32.21	37.53	40.24	41.3	40.65	38.386
45	HHO	40.22	38.54	40.46	39.92	35.95	39.018
46	Jaya	36.9	38.72	40.01	41	42.68	39.862
47	WOA	41.28	40.03	43.82	41.95	39.7	41.356
48	SCA	42.4	42.2	45.26	46.08	45.81	44.35
49	AOA	45.27	44.47	45.4	46.94	47.17	45.85
50	BFO	48.68	47.97	48.66	47.18	45.48	47.594

Table 7
The Wilcoxon signed-rank test results of the algorithms at a significance level of $\alpha = 0.05$.

vs.GGO		D=10	D=20	D=30	D=50	D=100
GO	+/=/−	1/11/0	3/9/0	6/23/0	6/23/0	10/15/4
GSK	+/=/−	2/10/0	9/3/0	9/19/1	13/14/2	19/10/0
MPA	+/=/−	7/5/0	6/5/1	15/13/1	19/10/0	28/1/0
SFS	+/=/−	5/7/0	3/9/0	14/14/1	24/5/0	26/3/0
SDO	+/=/−	4/8/0	7/5/0	24/5/0	29/0/0	28/1/0

in GGO, which enhances its ability to efficiently exploit promising regions of the search space—especially in complex, high-dimensional optimization problems.

All statistical comparisons were conducted at a significance level of $\alpha = 0.05$. The consistent outperformance of GGO under these rigorous tests confirms that the proposed algorithm offers stronger convergence behavior, greater global search reliability, and better scalability than existing methods, making it a robust and efficient choice for solving challenging high-dimensional optimization tasks.

5.2. Parallel speedup testing

To assess the computational efficiency of the proposed GPU-based implementation, we compare the parallel performance of GPU-GGO and its CPU-based counterpart using a set of widely adopted benchmark functions from the literature [58]. The characteristics of these test functions are summarized in Table 8, where "D" denotes the dimensionality of the search space, "U" refers to unimodal functions, "M" represents multimodal functions, "S" indicates separable problems, and "N" denotes non-separable problems. A common feature across all

Table 8
Parallel speedup test functions.

Number	Name	Equation	Feature	Bounds	Reference
F1	Sphere Variation	$f(x) = \sum_{i=1}^{D}(x_i - i)^2$	US	$[0, D]^D$	[59]
F2	Schwefel	$f(x) = \max_{i=1}^{D} \lvert x_i \rvert$	US	$[0, D]^D$	[58]
F3	Step	$f(x) = \sum_{i=1}^{D} \left(\lfloor x_i + 0.5 \rfloor \right)^2$	US	$[0, D]^D$	[58]
F4	Ackley	$f(x) = -20 \cdot \exp\left(-0.2 \cdot \sqrt{\frac{\sum_{i=1}^{D} x_i^2}{D}}\right) - \exp\left(\frac{\sum_{i=1}^{D} \cos(2\pi x_i)}{D}\right) + 20 + \exp(1)$	MN	$[0, D]^D$	[60]
F5	Rastrigin	$f(x) = 10D + \sum_{i=1}^{D} \left(x_i^2 - 10\cos(2\pi x_i)\right)$	MS	$[0, D]^D$	[61]
F6	Griewank	$f(x) = \frac{\sum_{i=1}^{D} x_i^2}{4000} - \prod_{i=1}^{D} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	MN	$[0, D]^D$	[62]
F7	Sphere	$f(x) = \sum_{i=1}^{D} x_i^2$	US	$[0, D]^D$	[59]

Table 9
Experimental testing hardware platform.

Name	Hardware parameters
CPU	Intel(R) Core(TM) i7 12800HX
GPU	NVIDIA GeForce RTX 4070 Laptop GPU
Operating system	Windows 11 based on X64 processor
Development environment	Microsoft Visual Studio 2022, CUDA 12.6

selected functions is that their theoretical global optimum is known and equals zero.

To ensure fairness and reproducibility, the maximum number of iterations is uniformly set to 5000 for all parallel tests. Each experiment is independently repeated 10 times, and the average performance is reported to mitigate the impact of random fluctuations and better reflect the algorithm's consistency.

It is important to note that the observed speedup is closely tied to the underlying hardware configuration. The specifications of the computational platform used for all parallel experiments are provided in Table 9.

5.2.1. Impact of particle size on performance

a. Effect of Particle Size on Accuracy

In Swarm Intelligence (SI) algorithms, the particle size refers to the number of individuals within the population. During the early stages of the search process, a larger population size typically promotes greater diversity, enabling broader exploration of the solution space and reducing the risk of premature convergence. For the Guided Growth Optimizer (GGO), increasing the number of particles can enhance classification precision, improve the efficacy of social learning, and bolster the algorithm's capacity to escape local optima. Fig. 10 illustrates the relationship between particle size and optimization accuracy in the GPU-accelerated version of GGO. The figure reports the best, average, and worst fitness values across 10 independent runs for each particle size setting.

As observed, increasing the particle size leads to improved convergence toward the global optimum. This improvement can be attributed to two factors: (1) the increased population diversity, which expands the algorithm's global search capabilities, and (2) the fine-grained parallelism in GGO's CUDA implementation, which mitigates the computational cost typically associated with larger populations. The use of high-quality random number generation via the `cuRAND` library, combined with dimension-level gradient guidance, ensures more thorough exploration and more accurate local refinement, thereby improving the algorithm's overall optimization performance.

b. Effect on Parallel Speedup

Speedup is a critical metric for evaluating the efficiency of parallel algorithms. It is formally defined by Eq. (17) as follows,

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \tag{17}$$

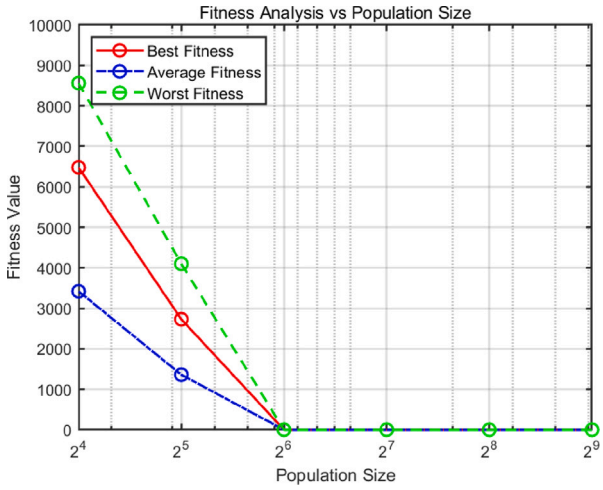


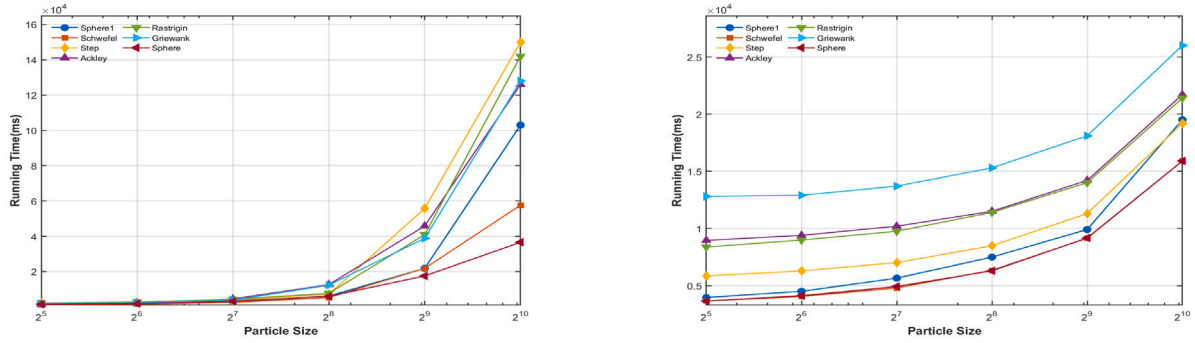
Fig. 10. Impact of particle size on optimization accuracy.

where T_{serial} and $T_{parallel}$ represent the execution times of the CPU-based serial and GPU-based parallel implementations, respectively.

Fig. 11 shows the relationship between particle size and execution time for both CPU and GPU implementations in 32-dimensional settings. At small particle sizes, the CPU-based implementation exhibits significantly shorter execution times than the GPU-based counterpart. This is primarily due to the fixed overhead associated with memory allocation, kernel launch latency, and CPU–GPU data communication, which becomes relatively dominant when the computational workload is low. These amortized overheads diminish the effective parallel efficiency of the GPU under small-scale workloads.

However, as the particle size increases, the benefits of GPU parallelism become more apparent. The runtime of the CPU-based serial implementation grows approximately linearly with the number of particles, while the GPU implementation exhibits a markedly slower growth rate. This behavior can be attributed to the unified memory and compute architecture of the GPU, wherein global data transfer between the CPU and GPU occurs only once during the execution phase. This design minimizes redundant memory access and communication costs, allowing computational gains to scale more effectively with problem size.

Fig. 12 illustrates the relationship between speedup and particle size in a 32-dimensional problem setting. As shown in the figure, several benchmark functions, such as the Rastrigin function, exhibit a near-linear increase in speedup as particle size grows. Among all tested functions, the Step function achieves the highest speedup, with performance reaching 7.8× acceleration at a particle count of 1024, with results reported as the mean value across 10 independent runs (7.8 ± 0.2), indicating stable and consistent performance. This substantial improvement is primarily due to GGO's hybrid coarse-grained and fine-grained parallelization strategy. In this architecture: Coarse-grained



(a) CPU running time (b) GPU running time

Fig. 11. The relationship between running time and particle size in 32 Dimensions.

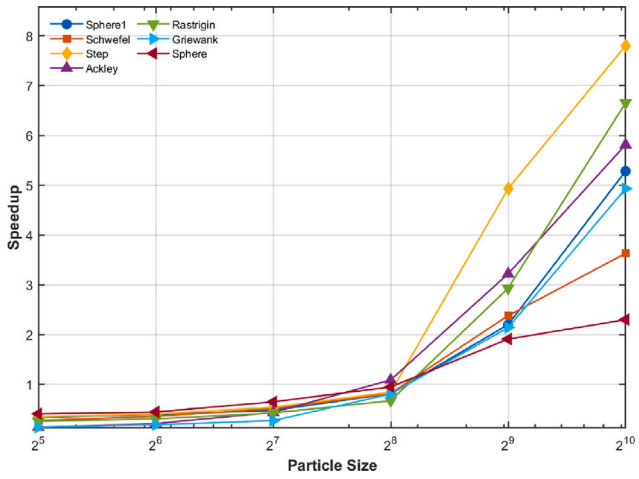


Fig. 12. The relationship between speedup and particle size in 32 dimensions.

parallelism manages population-level operations such as fitness evaluation and sorting; Fine-grained parallelism enables concurrent updates of particle dimensions using CUDA threads.

While fine-grained operations allow dimension-level concurrency, certain steps — such as sorting and global fitness ranking — still rely on coarse-grained synchronization and impose a lower bound on execution time. This interplay results in a speedup trend that is approximately linear, effectively reducing runtime without violating synchronization constraints.

Conversely, the Sphere function demonstrates the lowest observed speedup among all benchmark tests. Unlike more complex variants, the basic Sphere function involves only squaring each variable—a simple arithmetic operation with minimal memory access. This results in extremely low computational complexity, which reduces the relative benefit of parallel execution. As depicted in Fig. 11, while the GPU offers little room to optimize constant-time operations, the CPU benefits from the reduced workload and achieves faster performance. Consequently, the overall speedup is limited in such scenarios.

5.2.2. Impact of dimensionality on performance

a. Effect of Dimension on Optimization Accuracy

Problem dimensionality is a critical factor influencing the performance of optimization algorithms. As dimensionality increases, the complexity of the search space grows exponentially, often resulting in longer convergence times and reduced solution accuracy. Table 10 reports the optimization accuracy and computational speedup of the GPU-PSO and CPU-PSO implementations on the Sphere Variation function across various dimensions, with the particle size fixed at 512.

As seen in Table 10, the accuracy of the parallel implementation becomes increasingly sensitive to higher dimensionality. This degradation is primarily due to delayed information synchronization during the learning and reflection phases, which occurs only once per generation. Such delays may lead to inconsistencies in fitness propagation, causing occasional degradation in convergence accuracy and solution stability. In rare cases, suboptimal fitness values are observed, although their occurrence remains infrequent. Maintaining an appropriate ratio between population size and dimensionality can mitigate these effects. Specifically, increasing the number of particles in high-dimensional settings helps preserve population diversity and improves synchronization quality, thereby enhancing robustness and accuracy.

b. Effect of Dimensionality on Computational Speedup

Compared to population size, increasing dimensionality imposes a more substantial computational burden. While the proposed GGO algorithm has demonstrated superior performance over its serial counterpart in large-scale scenarios, it is essential to investigate how dimensionality affects parallel efficiency when the particle count is fixed.

Fig. 13 shows the relationship between runtime and dimensionality for both CPU and GPU implementations, with the number of particles set to 512. The CPU-based implementation shows a near-linear increase in execution time as dimensionality grows. In contrast, the GPU-based implementation exhibits low time complexity in low-to-moderate dimensions, but encounters a noticeable performance drop once the dimension exceeds a critical threshold. This performance bottleneck arises from register pressure in the fine-grained parallel strategy: as each particle's dimensionality increases, the per-thread register usage can exceed hardware limits, forcing the compiler to spill variables into slower local memory. This significantly hampers intra-thread efficiency, resulting in increased waiting times and reduced parallel scalability.

Fig. 14 depicts the corresponding speedup trends. Among the test functions, Sphere Variation demonstrates the highest speedup, approaching 7×, due to its simple computation pattern and minimal memory access. These characteristics enable the GPU to fully exploit its parallel architecture without inducing significant overhead. In contrast, other functions experience diminished speedup as dimensionality increases. This is mainly caused by register overflow and increased local memory usage, which reduce memory bandwidth and parallel efficiency. To address this issue, future improvements may include reducing excessive parameter dependencies or optimizing memory allocation strategies to better utilize high-speed shared resources.

In summary, parallel population-based intelligent algorithms exhibit notable speedup gains as problem scale increases. Although algorithmic design becomes more complex in higher dimensions — with challenges such as register overflow, memory bottlenecks, and the need for careful tuning of particle-to-dimension ratios — the computational acceleration remains substantial. These characteristics make parallel population-based intelligent algorithms highly valuable for real-time

Table 10
Optimization results and speedup of GPU-PSO vs. CPU-PSO on Sphere Variation under different dimensions.

Dimensions	Running results		Running times		Speedup
	CPU-PSO value	GPU-PSO value	CPU-PSO time	GPU-PSO time	
32	0	0	2.37E+04	9.91E+03	2.39E+00
64	2.21E−07	6.91E−07	3.67E+04	1.28E+04	2.87E+00
128	9.61E−01	3.90E+03	1.13E+05	2.07E+04	5.47E+00
256	7.24E+03	1.30E+06	2.60E+05	3.83E+04	6.78E+00

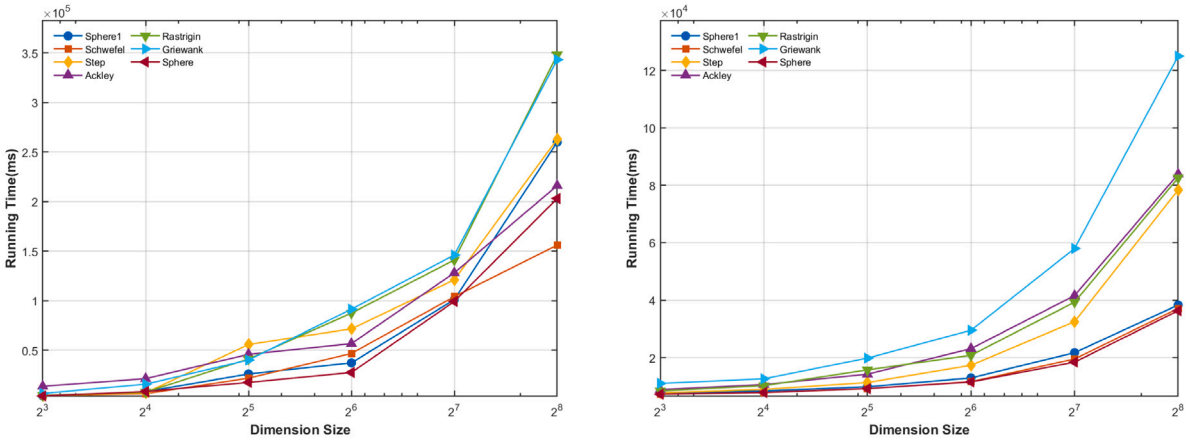


Fig. 13. Execution time vs. dimensionality for 512 particles.

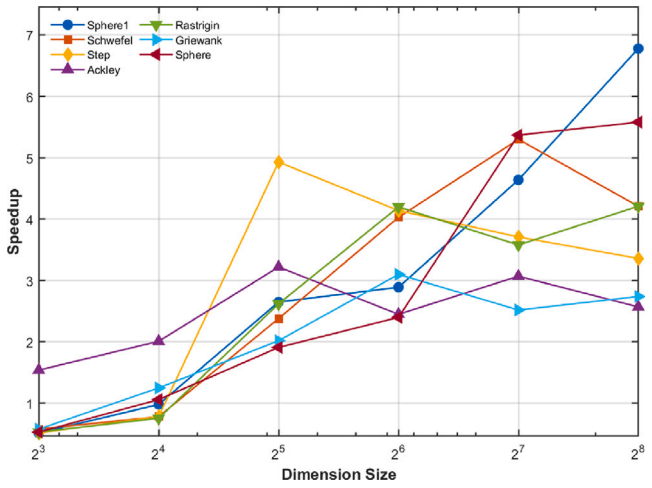


Fig. 14. The relationship between dimension size and speedup with 512 particles.

and large-scale optimization tasks, where high responsiveness and computational throughput are essential. Therefore, further exploration of scalable and memory-efficient parallel strategies remains a promising direction for future research in metaheuristic optimization.

6. Conclusions

This study presents the Gradient-Guided Growth Optimizer (GGO), a GPU-accelerated metaheuristic algorithm tailored for solving complex, high-dimensional numerical optimization problems. GGO builds upon the original Growth Optimizer (GO) by incorporating a novel, dimension-wise gradient-guided exploitation mechanism that enhances local search precision without requiring analytical gradients. This allows the algorithm to retain the robust global exploration ability of GO while significantly improving convergence accuracy in challenging optimization landscapes.

To exploit the computational power of modern hardware, GGO is implemented within a CUDA-based parallel computing framework that integrates both fine-grained and coarse-grained strategies. This hybrid design ensures efficient memory utilization, scalable execution, and low communication overhead, enabling high performance across diverse problem scales and dimensions.

Extensive evaluations on the CEC2017 and CEC2022 benchmark suites demonstrate that GGO consistently outperforms 49 state-of-the-art metaheuristic algorithms. In particular, GGO achieves up to 7.8× speedup compared to its CPU counterpart and attains top performance on 28 out of 29 high-dimensional functions, confirming its superior optimization capability and parallel scalability. Further analyses of population size and problem dimensionality provide insights into how algorithmic performance is influenced by parameter configurations and hardware constraints, offering practical guidance for real-world deployment.

In summary, GGO demonstrates a strong alignment with the goals of high-performance and parallel computing. Its robustness, efficiency, and adaptability make it a powerful optimization tool for applications in engineering design, machine learning, and scientific modeling.

Although the proposed GGO demonstrates strong performance and scalability, several promising directions remain for future research. First, algorithmic extensions could incorporate adaptive or self-tuning mechanisms to dynamically balance exploration and exploitation. Second, from a computational perspective, investigating heterogeneous architectures that integrate GPUs with multi-core CPUs, FPGAs, or cloud-based distributed platforms could further improve scalability. Third, conducting theoretical analyses of GGO's convergence properties and computational complexity would provide a stronger foundation for its general applicability. Moreover, extending the framework to multi-objective, constrained, or large-scale real-world optimization problems — such as engineering design, machine learning hyperparameter tuning, and scientific simulations — represents an important avenue. Finally, incorporating asynchronous communication and federated optimization strategies may enhance robustness in distributed or privacy-preserving environments. In summary, these future directions hold the

potential to further improve the applicability, efficiency, and impact of GGO across diverse scientific and engineering domains.

CRediT authorship contribution statement

Qingke Zhang: Writing – review & editing, Writing – original draft, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Wenliang Chen:** Writing – original draft, Visualization, Validation, Software, Investigation, Data curation. **Shuzhao Pang:** Validation, Software, Investigation. **Sichen Tao:** Validation, Resources, Investigation. **Conglin Li:** Visualization, Software, Formal analysis. **Xin Yin:** Validation, Software, Investigation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 62006144)

Data availability

Data will be made available on request.

References

[1] J. Tang, G. Liu, Q. Pan, A review on representative swarm intelligence algorithms for solving optimization problems: Applications and trends, IEEE/CAA J. Autom. Sin. 8 (10) (2021) 1627–1643.
[2] S. Mirjalili, S. Mirjalili, Genetic algorithm, in: Evolutionary Algorithms and Neural Networks: Theory and Applications, Springer, 2019, pp. 43–55.
[3] J. Kennedy, R. Eberhart, Particle swarm optimization, in: Proceedings of ICNN'95-International Conference on Neural Networks, vol. 4, ieee, 1995, pp. 1942–1948.
[4] M.F. Ahmad, N.A.M. Isa, W.H. Lim, K.M. Ang, Differential evolution: A recent review based on state-of-the-art works, Alex. Eng. J. 61 (5) (2022) 3831–3872.
[5] S. Fidanova, S. Fidanova, Ant Colony Optimization, Springer, 2021.
[6] E. Kaya, B. Gorkemli, B. Akay, D. Karaboga, A review on the studies employing artificial bee colony algorithm to solve combinatorial optimization problems, Eng. Appl. Artif. Intell. 115 (2022) 105311.
[7] A. Ali, A. AlShuaibi, M.W. Arshad, An integrated federated learning framework with optimization for industrial IoT intrusion detection, SHIFRA 2025 (2025) 110–117.
[8] V. Govindan, J. Jayaprakash, C. Park, J.R. Lee, I.N. Cangul, Optimization-based design and control of dynamic systems, Babylon. J. Math. 2023 (2023) 30–35.
[9] M.G. Yaseen, A.H. Ali, M. Alajanbi, An enhanced hybrid genetic-JAYA algorithm for feature selection and svm parameter optimization in intrusion detection systems: Evaluation on the CICIDS dataset, SHIFRA 2025 (2025) 98–109.
[10] Q. Zhang, H. Gao, Z.-H. Zhan, J. Li, H. Zhang, Growth optimizer: A powerful metaheuristic algorithm for solving continuous and discrete global optimization problems, Knowl.-Based Syst. 261 (2023) 110206.
[11] A. Dieuleveut, G. Fort, E. Moulines, H.-T. Wai, Stochastic approximation beyond gradient for signal processing and machine learning, IEEE Trans. Signal Process. 71 (2023) 3117–3148.
[12] G. Wu, R. Mallipeddi, P.N. Suganthan, Problem Definitions and Evaluation Criteria for the CEC 2017 Competition on Constrained Real-Parameter Optimization, Technical Report, National University of Defense Technology, Changsha, Hunan, PR China and Kyungpook National University, Daegu, South Korea and Nanyang Technological University, Singapore, 2017.
[13] J. Doe, J. Smith, A novel approach to evolutionary algorithms, in: 2022 IEEE Congress on Evolutionary Computation, CEC, IEEE, 2022, pp. 1234–1239, http://dx.doi.org/10.1109/CEC.2022.1234567.
[14] G. Ruetsch, M. Fatica, CUDA Fortran for Scientists and Engineers: Best Practices for Efficient CUDA Fortran Programming, Elsevier, 2024.
[15] A. Faramarzi, M. Heidarinejad, S. Mirjalili, A.H. Gandomi, Marine predators algorithm: A nature-inspired metaheuristic, Expert Syst. Appl. 155 (2020) 113377.
[16] W. Zhao, Z. Zhang, L. Wang, Supply-demand-based optimization: A novel economics-inspired algorithm for global optimization, IEEE Access 7 (2019) 73182–73206.
[17] T. Guilmeau, E. Chouzenoux, V. Elvira, Simulated annealing: A review and a new scheme, in: 2021 IEEE Statistical Signal Processing Workshop, SSP, IEEE, 2021, pp. 101–105.
[18] R.G. Reynolds, An introduction to cultural algorithms, in: Proceedings of the Third Annual Conference on Evolutionary Programming, vol. 24, World Scientific, 1994, pp. 131–139.
[19] M. Dubey, V. Kumar, M. Kaur, T.-P. Dao, A systematic review on harmony search algorithm: theory, literature, and applications, Math. Probl. Eng. 2021 (1) (2021) 5594267.
[20] J. Hu, H. Chen, A.A. Heidari, M. Wang, X. Zhang, Y. Chen, Z. Pan, Orthogonal learning covariance matrix for defects of grey wolf optimizer: Insights, balance, diversity, and feature selection, Knowl.-Based Syst. 213 (2021) 106684.
[21] Y. Liu, A.A. Heidari, Z. Cai, G. Liang, H. Chen, Z. Pan, A. Alsufyani, S. Bourouis, Simulated annealing-based dynamic step shuffled frog leaping algorithm: Optimal performance design and feature selection, Neurocomputing 503 (2022) 325–362.
[22] X. Zhang, S. Wen, D. Wang, Multi-population biogeography-based optimization algorithm and its application to image segmentation, Appl. Soft Comput. 124 (2022) 109005.
[23] X.-S. Yang, A. Slowik, Firefly algorithm, in: Swarm Intelligence Algorithms, CRC Press, 2020, pp. 163–174.
[24] M. Guerrero-Luis, F. Valdez, O. Castillo, A review on the cuckoo search algorithm, in: Fuzzy Logic Hybrid Extensions of Neural and Optimization Algorithms: Theory and Applications, Springer, 2021, pp. 113–124.
[25] Z. Younes, I. Alhamrouni, S. Mekhilef, M. Reyasudin, A memory-based gravitational search algorithm for solving economic dispatch problem in micro-grid, Ain Shams Eng. J. 12 (2) (2021) 1985–1994.
[26] T. Rahkar Farshi, M. Orujpour, A multi-modal bacterial foraging optimization algorithm, J. Ambient. Intell. Humaniz. Comput. 12 (11) (2021) 10035–10049.
[27] N. Bacanin, K. Alhazmi, M. Zivkovic, K. Venkatachalam, T. Bezdan, J. Nebhen, Training multi-layer perceptron with enhanced brain storm optimization metaheuristics, Comput. Mater. Contin. 70 (2) (2022) 4199–4215.
[28] G. Zhou, Y. Zhou, W. Deng, S. Yin, Y. Zhang, Advances in teaching–learning-based optimization algorithm: A comprehensive survey (ICIC2022), Neurocomputing 561 (2023) 126898.
[29] X. Quan, Y. Pang, J. Chen, X. Chu, L. Shangguan, Open pollution routing problem of logistics distribution in medical union based on differential search algorithm, Sci. Rep. 12 (1) (2022) 19472.
[30] S. Lalljith, I. Fleming, U. Pillay, K. Naicker, Z.J. Naidoo, A.K. Saha, Applications of flower pollination algorithm in electrical power systems: a review, IEEE Access 10 (2021) 8924–8947.
[31] M. Braik, M.H. Ryalat, H. Al-Zoubi, A novel meta-heuristic algorithm for solving numerical optimization problems: Ali baba and the forty thieves, Neural Comput. Appl. 34 (1) (2022) 409–455.
[32] S.N. Makhadmeh, M.A. Al-Betar, I.A. Doush, M.A. Awadallah, S. Kassaymeh, S. Mirjalili, R.A. Zitar, Recent advances in grey wolf optimizer, its versions and applications, Ieee Access 12 (2023) 22991–23028.
[33] S. Mirjalili, Moth-flame optimization algorithm: A novel nature-inspired heuristic paradigm, Knowl.-Based Syst. 89 (2015) 228–249.
[34] H. Shareef, A.A. Ibrahim, A.H. Mutlag, Lightning search algorithm, Appl. Soft Comput. 36 (2015) 315–333.
[35] H. Salimi, Stochastic fractal search: a powerful metaheuristic algorithm, Knowl.-Based Syst. 75 (2015) 1–18.
[36] M.H. Nadimi-Shahraki, H. Zamani, Z. Asghari Varzaneh, S. Mirjalili, A systematic review of the whale optimization algorithm: theoretical foundation, improvements, and hybridizations, Arch. Comput. Methods Eng. 30 (7) (2023) 4113–4159.
[37] R.V. Rao, Jaya: A simple and new optimization algorithm for solving constrained and unconstrained optimization problems, Int. J. Ind. Eng. Comput. 7 (1) (2016) 19–34.
[38] S. Mirjalili, SCA: a sine cosine algorithm for solving optimization problems, Knowl.-Based Syst. 96 (2016) 120–133.
[39] S. Mirjalili, S.M. Mirjalili, A. Hatamlou, Multi-verse optimizer: a nature-inspired algorithm for global optimization, Neural Comput. Appl. 27 (2) (2016) 495–513.
[40] S. Mirjalili, A.H. Gandomi, S.Z. Mirjalili, S. Saremi, H. Faris, S.M. Mirjalili, Salp swarm algorithm: A bio-inspired optimizer for engineering design problems, Adv. Eng. Softw. 114 (2017) 163–191.
[41] J. Pierezan, L.d.S. Coelho, Coyote optimization algorithm: A new metaheuristic for global optimization, in: 2018 IEEE Congress on Evolutionary Computation, CEC, IEEE, 2018, pp. 1–8.
[42] W. Zhao, L. Wang, Z. Zhang, Atom search optimization and its application to solve multi-objective engineering design problems, IEEE Access 7 (2019) 134929–134943.
[43] S. Yilmaz, S. Küçük, Artificial electric field algorithm for solving optimization problems, Appl. Soft Comput. 70 (2018) 337–355.
[44] A.A. Heidari, S. Mirjalili, H. Faris, I. Aljarah, M. Mafarja, H. Chen, Harris hawks optimization: Algorithm and applications, Future Gener. Comput. Syst. 97 (2019) 849–872.

[45] W. Zhao, Z. Zhang, L. Wang, Artificial ecosystem-based optimization: A novel nature-inspired metaheuristic algorithm, Neural Comput. Appl. 32 (13) (2020) 9385–9425.
[46] W. Zhao, Z. Zhang, L. Wang, Manta ray foraging optimization: An effective bio-inspired optimizer for engineering applications, Eng. Appl. Artif. Intell. 87 (2020) 103363.
[47] M. Abdel-Basset, R. Mohamed, D. El-Shahat, K.M. Sallam, Gaining sharing knowledge based algorithm for solving optimization problems, Appl. Soft Comput. 90 (2020) 106154.
[48] A. Faramarzi, M. Heidarinejad, B. Stephens, S. Mirjalili, Equilibrium optimizer: A novel optimization algorithm, Knowl.-Based Syst. 191 (2020) 105190.
[49] M. Abdel-Basset, R. Mohamed, D. El-Shahat, K.M. Sallam, Coronavirus herd immunity optimizer (CHIO), Knowl.-Based Syst. 200 (2020) 106055.
[50] Q. Askari, M. Saeed, I. Younas, Heap-based optimizer inspired by corporate rank hierarchy for global optimization, Expert Syst. Appl. 161 (2020) 113662.
[51] L. Abualigah, A. Diabat, S. Mirjalili, M.A. Elaziz, A.H. Gandomi, Arithmetic optimization algorithm: A new metaheuristic for solving global optimization problems, Expert Syst. Appl. 166 (2021) 114009.
[52] M. Abdel-Basset, R. Mohamed, D. El-Shahat, K.M. Sallam, Runge-kutta optimizer: A new metaheuristic optimization algorithm for solving complex optimization problems, Knowl.-Based Syst. 218 (2021) 106838.
[53] M. Braik, A. Sheta, H. Al-Hiary, White shark optimizer: A novel bio-inspired metaheuristic algorithm for global optimization, Knowl.-Based Syst. 215 (2021) 106755.
[54] Q. Askari, M. Saeed, I. Younas, Honey badger algorithm: A novel bio-inspired optimization algorithm, Expert Syst. Appl. 176 (2021) 114788.
[55] M. Abdel-Basset, R. Mohamed, D. El-Shahat, K.M. Sallam, Dwarf mongoose optimization algorithm, Expert Syst. Appl. 191 (2022) 116283.
[56] M. Abdel-Basset, R. Mohamed, D. El-Shahat, K.M. Sallam, Snake optimizer: A novel metaheuristic optimization algorithm, Knowl.-Based Syst. 221 (2022) 106937.
[57] M. Abdel-Basset, R. Mohamed, D. El-Shahat, K.M. Sallam, Improved growth optimizer: A novel metaheuristic optimization algorithm, Expert Syst. Appl. 191 (2022) 116285.
[58] S.-Y. Ho, L.-S. Shu, J.-H. Chen, Intelligent evolutionary algorithms for large parameter optimization problems, IEEE Trans. Evol. Comput. 8 (6) (2004) 522–541.
[59] M. Molga, C. Smutnicki, Test functions for optimization needs, Test Funct. Optim. Needs 101 (48) (2005) 32.
[60] M.A. Potter, K.A. De Jong, A cooperative coevolutionary approach to function optimization, in: International Conference on Parallel Problem Solving from Nature, Springer, 1994, pp. 249–257.
[61] H. Mühlenbein, M. Schomisch, J. Born, The parallel genetic algorithm as function optimizer, Parallel Comput. 17 (6–7) (1991) 619–632.
[62] A.O. Griewank, Generalized descent for global optimization, J. Optim. Theory Appl. 34 (1981) 11–39.