Applied Soft Computing 144 (2023) 110499

Contents lists available at ScienceDirect

Applied Soft Computing

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

A parallel particle swarm optimization algorithm based on GPU/CUDA

Yanhong Zhuo [a,*], Tao Zhang [a], Feng Du [b], Ruilin Liu [a]

[a] School of Information and Mathematics, Yangtze University, Jingzhou, Hubei, China
[b] School of Mathematics and Physics, Jingchu University of Technology, JingMen, Hubei, China

ARTICLE INFO

Article history:
Received 18 June 2022
Received in revised form 25 May 2023
Accepted 6 June 2023
Available online 12 June 2023

Keywords:
Particle swarm optimization algorithm
Parallel computing
CUDA
GPU

ABSTRACT

Parallel computing is the main way to improve the computational efficiency of metaheuristic algorithms for solving high-dimensional, nonlinear optimization problems. Previous studies have typically only implemented local parallelism for the particle swarm optimization (PSO) algorithm. In this study, we proposed a new parallel particle swarm optimization algorithm (GPU-PSO) based on the Graphics Processing Units (GPU) and Compute Unified Device Architecture (CUDA), which uses a combination of coarse-grained parallelism and fine-grained parallelism to achieve global parallelism. In addition, we designed a data structure based on CUDA features and utilized a merged memory access mode to further improve data-parallel processing and data access efficiency. Experimental results show that the algorithm effectively reduces the solution time of PSO for solving high-dimensional, large-scale optimization problems. The speedup ratio increases with the dimensionality of the objective function, where the speedup ratio is up to 2000 times for the high-dimensional Ackley function.

1. Introduction

Meta-heuristic algorithms have the feature of searching globally to find approximate solutions to optimal solutions, which are widely used to solve many optimization problems in real such as combinatorial optimization [1], robot path planning [2], function optimization [3], traveling salesman problem [4], wireless sensor network performance optimization [5], and others [6]. With the rapid development of modern science and technology, many practical numerical optimization problems have become extremely complex, which often have non-linear, high-dimensional objective functions. However, most meta-heuristic algorithms lack scalability. When encountering high-dimensional and complex problems, the performance decreases in terms of both time complexity and effectiveness.

Due to significant improvements in hardware devices and networking technologies, parallelization of model simulations (as opposed to serial computing) has become an effective strategy to reduce execution time and improve the quality of the solutions found. One of the major issues for metaheuristics is to rethink existing parallel models and programming paradigms to allow their deployment on GPU accelerators. One of the most important approaches to parallel computing is to divide a heavy computational task into multiple independent loads so that subtasks are assigned to multiple processors simultaneously. The CUDA platform supported by the General Purpose Graphics Processing Unit (GPGPU) allows parts of an algorithm implementation to be distributed across up to thousands of graphics processing cores on one or more platforms, with the GPU specifically designed to handle floating-point operations and intensive parallel execution (see Table 1). Parallelization of various heuristic algorithms [7–10] has been shown to be effective including the known particle swarm optimization algorithm, which can speed up the solution time of high-dimensional optimization problem models.

Since Veronses [11] ported particle swarm optimization algorithms to the GPU platform in 2009, different forms of GPU-PSO have emerged. Parallel particle swarm optimization algorithms can be classified as coarse-grained parallelism and fine-grained parallelism depending on the degree of data parallelism. In CUDA thread allocation, a coarse-grained parallelism model is an approach that utilizes the correspondence of particles to threads. For example, Y. Zhou and Y. Tan [12] proposed a method for parallel SPSO based on a GPU. The running speed of GPU-SPSO can be more than 11 times as fast as that of CPU-SPSO, with the same performance. Mussia proposed two different PSO algorithms that utilize GPU parallelism [13]. The execution speed of two parallel SPSO algorithms was compared with sequential SPSO algorithms on benchmark functions. Hung and Wang [14] proposed a GPU-accelerated PSO (GPSO) algorithm by using a thread pool model and implementing GPSO on a GPU. Numerical results show that when solving the 100-dimensional test problems with 65,536 particles, GPSO has achieved up to 83X speedups. Fine-grained parallelism improves parallelism efficiency by adopting a one-to-one correspondence between particles and thread blocks and

* Corresponding author.
E-mail addresses: yhZhuo@yangtzeu.edu.cn (Y. Zhuo), tzhang@yangtzeu.edu.cn (T. Zhang), defengdu123@163.com (F. Du), lrlgogo@163.com (R. Liu).

https://doi.org/10.1016/j.asoc.2023.110499
1568-4946/

Table 1
Abbreviation table.

Abbreviation	Full name	Abbreviation	Full name
SPSO	standard Particle Swarm Optimization	CUDA	Compute Unified Device Architecture
CPU	Central Processing Unit	GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit	VRAM	Video Random Access Memory
SIMD	Single Instruction Multi Data	SIMT	Single Instruction Multiple Threads
ALU	Arithmetic Logic Unit	CU	Control Unit and Cache Unit

dimensions and threads in CUDA thread allocation. For example, Calazan proposed a GPU-based [15] high-dimensional parallel particle swarm optimization algorithm in a fine-grained parallel mode. Hussain [16] utilizes merged memory accesses on a CUDA architecture to parallelize a standard particle swarm optimization algorithm on a GPU, running nearly 46 times faster than a parallel particle swarm optimization algorithm on a CPU. Silva and Filho [17] proposed to use multiple sub-populations in one-to-one correspondence with the GPU's thread blocks, which makes data parallelism more efficient and avoids instruction fragmentation. They also provide two communication mechanisms and two topologies that allow subpopulations to exchange information and collaborate via GPU global memory.

Nevertheless, previous studies have typically only implemented local parallelism for the entire algorithm, resulting in too much time spent on communication between its CPU and GPU. For parallel models based on a combination of coarse-grained and fine-grained parallelism, it is a problem worth investigating how to achieve global parallelism and reduce CPU–GPU communication. To maximize parallel performance, we believe that an optimal GPU-based parallel algorithm should have the following attributes: (a) a load-balancing strategy is always used in the preparation of the parallel scheme; (b) the corresponding parallel scheme is selected for different operational sessions to obtain optimal efficiency; (c) CPU–GPU communication is reduced so that all computational steps are implemented in parallel on the GPU to achieve the global parallel effect.

Along this vein, this study proposes a new parallel particle swarm optimization algorithm based on GPU and CUDA, which uses a combination of coarse-grained parallelism and fine-grained parallelism to achieve global parallelism. To further improve the computational efficiency of the approach, two optimization strategies are utilized in our algorithm design, the CUDA feature-based data structure and the merged memory access model. The parallel statute ordering method, which has one of the lowest average times of any sorting algorithm, is used in the process of updating the global individual optimum. Experiments on 10 standard test functions show that the GPU-PSO algorithm has the same numerical solution as the serial CPU-PSO algorithm after every iteration update and that its speedup ratio is correlated linearly with the dimensions of the objective function and the population size. In particular, the maximum speedup ratio on the Ackley function is as high as 2000 times.

The rest of this article is organized as follows. In Section 2, we briefly introduce some basic concepts and models, including, GPU computing, CUDA architecture, and consolidated memory access. After that, we introduce the basic concepts and models of the traditional PSO algorithm and the implementation of serial PSO based on the CPU. In Section 4, we provide the design and implementation principle of the improved GPU-PSO algorithm based on CPU-PSO. Subsequently, in Section 5, the experimental results obtained are presented, analyzed and compared with previous implementations in terms of execution time, speedup, and fitness values. Finally, in Section 6, we make some conclusions and point out the direction of future work.

2. Architecture background

2.1. GPU

GPUs were originally designed specifically for computer graphics and graphics processing. GPU has many advantages due to its special architecture [18]. The architecture comparison of CPU and GPU is shown in Fig. 1. First, GPUs have less CU than CPUs, but more ALUs, making GPUs particularly suitable for intensive computing tasks. Second, the GPU uses more transistors for data processing rather than data caching and flow control, which allows it to perform more floating-point operations per second than the CPU. Finally, GPU belongs to the SIMD class in Flynn classification [19], in which the simultaneous operation of multiple data sets is controlled by the same set of instructions at the same time.

In recent years, driven by the special effects industry in games and movies, GPU technology has developed rapidly, and its powerful foothold in computing power has gradually emerged. As it meets the need to handle big data and intensive repetitive operations, experts outside the graphics field are beginning to experiment with GPUs for general-purpose computing. GPU has been successfully applied to computer vision problems [20], Voronoi graph [21], neural network computing [22], and other computing fields. GPUs have become the mainstream of computing. Many platforms and programming models for GPU computing have been proposed over the years, with the most important platforms being CUDA [23], OpenCL [24], and MPI [25].

2.2. CUDA

NVIDIA proposes CUDA, which makes GPUs one of the main focuses for parallel processing by adding easy-to-use programmable interfaces and providing a programming language for CUDA-enabled GPUs. In CUDA architecture, there are two types of parallel processing, namely data parallelism, and task parallelism. Data parallelism refers to the parallel execution between different threads started by a single kernel task. Because the processing logic of all threads corresponding to the same kernel function is the same [26]. But each thread has different data. CUDA employs a SIMT model in which each thread of a block executes the same set of instructions on different data streams (in this application, different particles and particle sizes for subgroups). During CUDA parallel application implementation, blocks are represented by kernel functions. To run on the GPU, each kernel function is organized into blocks and threads. When a CUDA program running on the host CPU invokes the kernel grid, the grid's blocks are enumerated and distributed across multiple processors with available execution power. It uses a unique architecture called SIMT to manage such a large number of threads. A thread block's threads can run concurrently. The thread ID identifies each thread, which can be deduced by the following formula according to the 'CUDA C++ Programming Guide' and as shown in Fig. 2.

$$\begin{aligned} ID =& (blockIdx.y * gridDim.x + blockIdx.x) \\ & * (blockDim.x * blockDim.y) \\ & + threadIdx.y * blockDim.x + threadIdx.x \end{aligned} \tag{1}$$

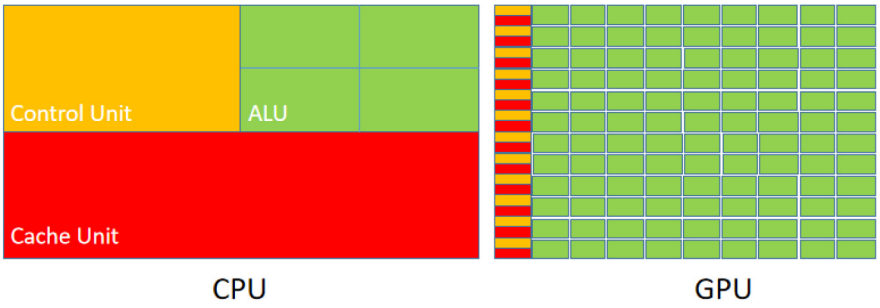


Fig. 1. The architecture comparison of CPU and GPU.

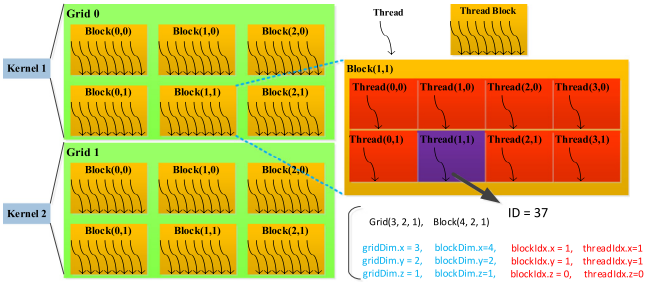


Fig. 2. The thread batching of a kernel function in CUDA and a example of thread ID.

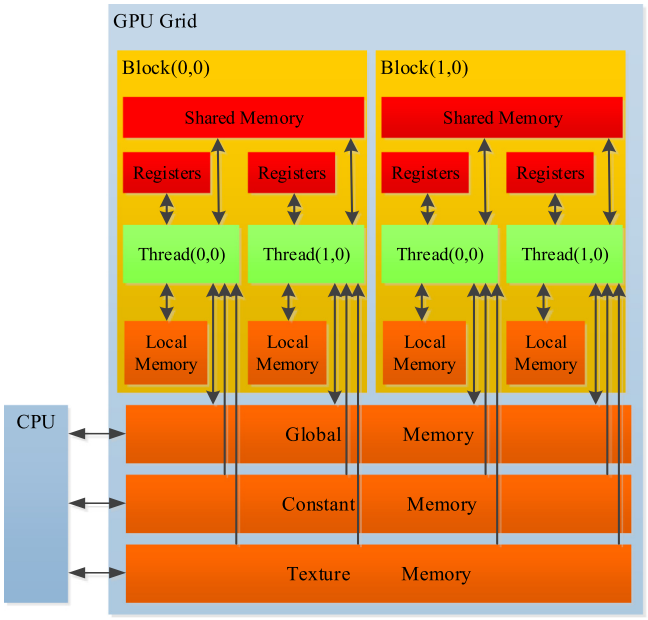


Fig. 3. Memory hierarchy of CUDA/GPU.

Some applications have been developed based on CUDA, such as matrix multiplication [27], large array parallel prefix sum [28], real-time visual hull calculation [29], image denoising [30], and others. Readers interested in learning more about CUDA programming can visit the NVIDIA CUDA ZONE website. Because the programming language used in this study is Python, the GPU-based parallel PSO algorithm is primarily realized using PyCUDA.

2.3. Coalescing memory access

CUDA/GPU has their memory hierarchy. Currently, available memory is global memory, shared memory, local memory, constant memory, and registers. GPUs provide memory with high bandwidth as shown in Fig. 3. Global Memory is equivalent to GPU video memory in a sense, and the kernel function reads and writes video memory through global memory. The Global Memory is the only source of input data and write results to the kernel function. Shared memory is located on the GPU chip and has access latency second only to registers. Shared Memory can be accessed by all threads in the same thread block, thus reducing the communication overhead between different threads in the thread block. Local storage is an address space that is virtual from global memory and has no specific storage unit in hardware. Local Memory is designed to hold large single-threaded arrays and variables, and is specifically designed for situations where registers cannot meet storage requirements. Local Memory is thread private and not visible between threads. Since GPU hardware has no specific memory unit for local storage, access to it is slow. The register is the fastest memory unit located on the SM of the GPU chip for storing local variables. When a kernel function is started, all the registers in each SM(SMX) are allocated to the specified thread.

It is critical to follow the correct memory access pattern in order to obtain the maximum memory bandwidth. Merging the memory access mode is the most efficient way to access global memory on a GPU. Memory consolidation takes advantage of the fact that all threads in a warp execute the same instructions at any given point in time, so the optimal memory access pattern is achieved when all threads in the warp are contiguous when accessing global memory. That is, the most favorable access pattern can be achieved when parallel threads running the same instructions access contiguous locations in global storage [16]. Alternatively, when simultaneous memory access for warp threads is not consolidated, VRAM bandwidth becomes inefficient and time-consuming. The thread in this case accesses at least two memory segments in the warp. When multiple threads access different segments of VRAM, bandwidth efficiency suffers significantly.

3. CPU-PSO algorithm

The PSO algorithm proposed by Dr. Kennedy and Professor Eberhart in 1995 [31] is a random heuristic algorithm with inherent parallelism and is implemented easily. This algorithm can deal with nonlinear optimization functions that are not required differentiable, differentiable, or continuous properties, and have a fast convergence rate. In the PSO algorithm, each particle represents a potential solution, and all particles from a population. In the solution space, particles determine the velocity and direction of optimization according to their historical information and group information, to find the optimal solution. The function of single-objective optimization to be minimized (/maximized) is equationted as

$$min\ f\ (X)\ \ s.t.\ \ X \in S \subseteq R^D \tag{2}$$

Where $X = (X_1, X_2, \ldots, X_N)$ is a decision variable matrix comprised of N vectors with m dimensions in the feasible space S.

Each dimension of an individual X_i of an initial population is randomly generated:

$$X_{ij} = Low_j + rand(0,1) * (Up_j - Low_j), \quad i = 1, 2, \ldots, N, \quad j = 1, 2, \ldots, D \tag{3}$$

where N and D are the population size and the dimension of the individual respectively. $rand(0, 1)$ represents a random number generated uniformly in the range [0, 1], and Up_j and Low_j are the upper and lower bounds of the jth dimension of the individual, respectively.

The velocity and position update equations of the algorithm as following.

$$V_i^{(t+1)} = \omega V_i^{(t)} + c_1 \cdot r_1 \cdot \left(\hat{P}_i^{(t)} - X_i^{(t)}\right) + c_2 \cdot r_2 \cdot \left(\hat{G}_i^{(t)} - X_i^{(t)}\right) \tag{4}$$

$$X_i^{(t+1)} = X_i^{(t)} + V_i^{(t+1)} \tag{5}$$

Where, ω is the inertial weight and lying between 0 and 1, which is used to balance the global exploration ability and local development ability of particles. c_1 and c_2 are their learning factors with respect to individuals, non-negative constants, r_1 and r_2 are their learning factors about group society, which are uniformly distributed random numbers between 0 and 1, $X_i = (X_{i1}, X_{i2}, \ldots, X_{iD})$ is the position of the ith particle in the D-dimensional search space, $X_{id} \in [-X_{max}, X_{max}]$, X_{max} limits the scope of particle search space, $V_i = (V_{i1}, V_{i2}, \ldots, V_{iD})$ is the velocity of the ith particle, $V_{id} \in [-V_{max}, V_{max}]$, V_{max} is the maximum velocity constraint for particle flight, $\hat{P}_i = \left(\hat{P}_{i1}, \hat{P}_{i2}, \ldots, \hat{P}_{iD}\right)^T$ is the personal best position of ith particle, $\hat{G}_i = \left(\hat{G}_{i1}, \hat{G}_{i2}, \ldots, \hat{G}_{iD}\right)^T$ is the global best position of whole swarm.

A greedy selection mechanism is used to update the individuals of the current population to generate the next population. For a given fitness function, if the fitness value of the trial individual $\hat{p}_i$ is better than that of X_i, $\hat{P}_i$ is retained; otherwise, x_i is retained. For a minimization problem, the two process for update pbest and gbest can be equationted at iteration $t + 1$ as follows

$$\begin{cases} P_i^* = f(\hat{P}_i^{(t)}) \ \ and \ \ \hat{P}_i^{(t+1)} = \hat{P}_i^{(t)}, & \text{if} \quad f(X_i^{(t+1)}) \geq f(\hat{P}_i^{(t)}) \\ P_i^* = f(X_i^{(t+1)}) \ \ and \ \ \hat{P}_i^{(t+1)} = X_i^{(t+1)}, & \text{if} \quad f(X_i^{(t+1)}) < f(\hat{P}_i^{(t)}) \end{cases} \tag{6}$$

$$\hat{G}^{(t+1)} = X_k \in \left\{\hat{P}_1^{(t+1)}, \hat{P}_2^{(t+1)}, \ldots, \hat{P}_N^{(t+1)}\right\} \tag{7}$$

$$G^{*(t+1)} = f\left(\hat{G}^{(t+1)}\right) \tag{8}$$

Where, P_i^* is the personal optimal fitness for ith particle and G^* is the global optimal fitness. Especially, $f(x_k) = min\left\{f(\hat{P}_1^{(t+1)}), f(\hat{P}_2^{(t+1)}), \ldots, f(\hat{P}_N^{(t+1)})\right\}$, and $f(\cdot)$ is the objective function of the minimization problem.

Assuming that the problem dimension in this paper is M and the population size is N, the definition of initialization variables is shown in Table 2. Currently, most implementations of the PSO algorithm are executed sequentially on the CPU, also known as CPU-PSO. The algorithm flows of CPU-PSO are described in Algorithm 1.

Algorithm 1: The pseudo code of CPU-PSO

Input: $N, D, Up_j, Low_j, X_{max}, V_{min}, V_{max}$;
Output: $G^*, \hat{G}$;
Initialize population individual position and describe it as $X \in \left(Low_j, Up_j\right)$ by Eq. (3);
Randomly initialize the individual search direction of the population and describe it as $V \in (V_{min}, V_{max})$;
Initialize population personal best position denoted as $\hat{P}$ which equates the X;
while *the number of iterations is less than the maximum number of iterations* **do**
  Evaluate individuals fitness value and document the best individual;
  Velocity update operation is performed by Eq. (4);
  Position update operations is performed by Eq. (5);
  Update personal optimal fitness and position by Eq. (6);
  Update global fitness and position by Eqs. (7) and (8);
end

4. Motivation

With the significant improvement of hardware equipment and network technology, the parallel implementation of evolutionary algorithms has become an effective strategy to reduce the time cost. At present, parallel algorithms are in the development stage, and there is no general design method. For the design of parallel algorithms, the three most widely used design strategies are as follows: (1) check and exploit the parallelism contained in the existing serial algorithms, and directly parallelize them. While this approach is not entirely appropriate for all problems, it is useful for many application problems. (2) Starting from the description of the problem, a new parallel method is redesigned according to the basic characteristics of the problem. Although this approach is somewhat challenging, the designed parallel algorithms are generally efficient. (3) Use current parallel algorithms to help you solve new problems.

The PSO algorithm is a classical meta-heuristic algorithm with inherent parallelism. According to the existing reference materials of parallel PSO algorithm based on CUDA platform, it is found that the parallelism degree and the communication time between CPU and GPU are the main factors affecting the computing time. The degree of parallelism can be divided into global parallelism and local parallelism, and fine-grained parallelism and coarse-grained parallelism in specific calculation steps. Therefore, when using PSO algorithm to solve the optimization problem, only by realizing global parallelism, selecting the most appropriate parallel mode for each computing plate, and controlling the communication cost can the time cost be reduced to the greatest extent.

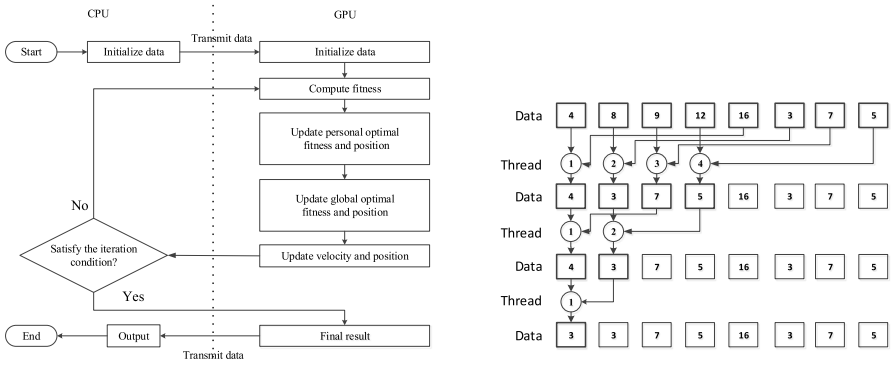
Inspired by the above discussion, GPU-PSO is proposed to realize global parallelism and control the CPU and GPU to communicate only once, using a combination of coarse and fine grain parallel computing. By making full use of the parallel computing capability of GPU, we hope that the GPU-PSO algorithm can obtain the same solution as the CPU-PSO algorithm when solving high-dimensional, large population optimization problems and improve the running speed.

5. GPU-PSO algorithm

The main flow of GPU-PSO is shown in Fig. 4(a). Obviously, on the heterogeneous platform with GPU and CPU working together, the CPU is the leader of the whole algorithm, responsible for the control of the algorithm process and the initialization of the data. The GPU execution part mainly completes the parallel calculation of the fitness value and the population update in the

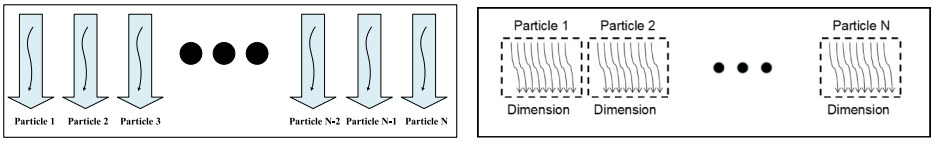
Table 2
Variable setting.

Variable name	Shape	Variable name	Shape
Position: X	$M \times N$ matrix	Velocity: V	$M \times N$ matrix
Fitness Value: F	$N \times 1$ variable	Parameter: $\omega, c1, c2, r1, r2$	$M \times N$ matrix
Personal optimal fitness value: P^*	$N \times 1$ variable	Personal optimal position: $\hat{P}$	$M \times N$ matrix
Global optimal fitness value: G^*	1×1 variable	Global optimal position: $\hat{G}$	$M \times N$ matrix



(a) The main flows of GPU-PSO algorithm (b) An example of merge sort

Fig. 4. (a) The main flows of GPU-PSO algorithm, and (b) An example of merge sort for updating global best fitness.



(a) coarse-grained parallelism (b) fine-grained parallelism

Fig. 5. The data parallel mode includes (a) coarse-grained parallelism and (b) fine-grained parallelism.

PSO algorithm. They cooperate to complete the whole algorithm process.

The computational processes executed on the GPU all use a parallel computing model based on data parallelism in the CUDA architecture. The parallel strategy of the GPU-PSO algorithm includes coarse-grained parallelism and fine-grained parallelism. The coarse-grained parallel strategy is that one thread performs the computational operation of one particle alone, and the parallel computation of the particle swarm is processed by a block of threads composed of multiple threads. Fig. 5(a) represents N particles executing the same operation instruction at the same time, and their execution time is equal to the execution time of a single thread. In the GPU-PSO algorithm, the process of updating individual optimal fitness values uses a coarse-grained parallel strategy. The fine-grain parallel strategy is that one thread corresponds to one dimension of a single particle, and the parallel computation of each particle is handled by a block of threads. The fine-grain parallelism model is shown in Fig. 5(b), where each dimension of N particles performs exactly the same instruction operation. In this case, the black curve represents a thread that corresponds to the data of a single particle with elements on a single dimension. The black dashed box represents a thread block, and the threads within a thread block perform the same operation instructions. Therefore, multiple thread blocks are required when a generation of particle updates is completed. In the GPU-PSO algorithm, the update process of velocity and position, as well as the search process of individual optimal positions are mainly used in fine-grained parallel strategies. According to the characteristics of various parallel strategies, two parallel strategies are used interactively in the calculation process of the fitness function values.

In the PSO algorithm, when calculating the fitness value of a function, the objective function often involves some accumulation and summation operations. In the serial algorithm, considering the computing characteristics of the computer, when the value of the accumulation is large enough and needs to be accumulated. The value of is relatively small, there will be an easy-to-ignore the situation: $\sum a_i + a_n = a_i \, (n \to \infty)$. Moreover, protocol sort can save more time than bubble sort, which is often used in serial algorithm, when judging the global optimum. Therefore, in the GPU-PSO algorithm, we will use the parallel protocol to improve operational efficiency and solution accuracy, as shown in Fig. 4(b). The summation pattern is similar, only the calculation rules between adjacent threads need to be changed.

The core of the GPU- PSO algorithm is the design of the GPU kernel function for parallel execution, and the details of our designed kernel function and other auxiliary processes in the algorithm are described below.

Step 1: Initialization

Due to the great difference in different storage characteristics in CUDA multi-level storage systems, reasonable allocation and use of threads is the key to improving algorithm parallelism. The CPU can then copy the initial data into device memory via the *memcpy_htod*() function. Such data replication takes a long time, so this process will not be considered when comparing the computing performance of GPU-PSO and CPU-PSO.

Step 2: Compute Fitness Values of Particles

The calculation of the fitness value of a single objective function value involves more high-density arithmetic calculation,

which is the more important task in the whole search process. If this process is parallelized, the overall performance of GPU-PSO will be improved. Therefore, this paper combines coarse-grained and fine-grained parallel methods in this process. In this process, fine-grained parallelism means that each element in the matrix performs the same logical computation, for which the number of threads that need to be opened in CUDA is $M \times N$. Suppose its calculation model is $Y = Z(X)$, where X is the independent variable, $Z(X)$ is a function, Y is the dependent variable, and X and Y are in the same matrix form. Coarse-grained parallelism refers to the same logical calculation of single row elements in the independent variable matrix, and its calculation model can be expressed as $F = H(E)$. Where, the independent variable E is matrix form, $H(E)$ represents a function, and the dependent variable F is vector form. For this, CUDA needs to open N threads and only loop M times. The algorithm for fitness values computing of all the particles is shown in Algorithm 2.

Algorithm 2: Compute Fitness Values

Input: X_{ij}^t
Output: F_i^t
Set the 'block size' and 'grid size', tx is the threads number obtained by Eq. (1);
if $tx < M \times N$ **then**
  $h_{tx} := Z\left(X_{tx}^t\right)$, Z is a function which parallel compute every element;
end
for $j := 1$ *to* M **do**
  if $tx < N$ **then**
    $F_{tx}^t := H(h_{tx})$, H is a function which parallel compute every particle;
  end
end

Step 3: Update Pbest

By updating the fitness values, each particle may be in a better position than ever before. For the current state of particle swarm, Algorithm 3 can be used to update the individual optimal fitness value vector and individual optimal position matrix according to Eq. (6). The time complexity of implementing the process is $\mathcal{O}(1)$.

Step 4: Update Gbest

In this process, Algorithm 4 is designed by combining the idea of the parallel comparison of protocols (see Fig. 4(b)). Due to the merge sort mode adopted by GPU-PSO, the time complexity is $\mathcal{O}\left(nlog_2^n\right)$ and space complexity is $\mathcal{O}(n/2)$ regarding updating the global optimal fitness value. The bubble sorting method adopted by CPU-PSO has the time complexity of $\mathcal{O}(n)$ and space complexity of $\mathcal{O}(1)$.

Step 5: Update Velocity and Position

After updating the pbest and gbest, each particle will lock its new search direction and update velocity and position. According to Eq. (4) and (5), it is found that this process can adopt fine-grained parallel mode, which needs to open $N \times M$ threads. The velocity and position update strategy of all the particles in the GPU-PSO algorithm is shown in Algorithm 5. The time complexity of implementing the process is $\mathcal{O}(1)$.

6. Simulation experiments

To compare the performance of GPU-PSO and CPU-PSO, this paper selects 10 internationally common benchmark test functions from literature [32], as shown in Table 3. And these functions have same the global optimal theory value which is 0. In this table, 'U' represents a single-peak function, 'M' represents a multi-peak function, 'S' represents independent, and 'N' represents dependent.

Algorithm 3: Update Personal Optimal Fitness Value and Position

Input: X_{ij}^{t+1}, $\hat{P}_i^t$, P_{ti}^*, F_i^{t+1}
Output: $\hat{P}$, P^*
Set the 'block size' and 'grid size', tx is the threads number obtained by Eq. (1);
\\ Padding N-dimensional vectors $\hat{P}^\star$ into $(M \times N)$-dimensional matrices $\widetilde{P}$;
\\ Padding N-dimensional vectors F into $(M \times N)$-dimensional matrices $\widetilde{F}$;
for $j := 1$ *to* M **do**
  for $tx := 0$ *to* N *parallel* **do**
    $\widetilde{P_{txj}} := \hat{P}_{tx}^*$
    $\widetilde{F_{txj}} := F_{tx}$
  end
end
\\ Compare $\widetilde{F}$ and $\widetilde{P}$ in parallel;
for $tx := 0$ *to* $(M \times N)$ *parallel* **do**
  if $\widetilde{F_{tx}} < \widetilde{P_{tx}}$ **then**
    $\widetilde{P_{tx}} := \widetilde{F_{tx}}$
    $\hat{P}_{tx}^t := X_{tx}^t$
  end
end
return P^*, $\hat{P}$;

Algorithm 4: Update Global Optimal Fitness Value

Input: P^*, $K = \lceil log_2^N \rceil + 1$, N is the number of particles;
Output: G^*, global optimal fitness value matrix;
Set the 'block size' and 'grid size', tx is the threads number obtained by Eq. (1);
$P^0 := P^*$,
for $k := 1$ *to* K **do**
  $b := 2^k / (2 \cdot k)$
  for $tx := 0$ *to* 2^{k-1} *in parallel* **do**
    $A_{tx} := P_{tx}^0$
    $B_{tx} := P_{tx+b}^*$
    if $A_{tx} > B_{tx}$ **then**
      $P_{tx}^0 := B_{tx}$
    end
  end
  $G^* := P_0^0$
end
return G^*

Algorithm 5: Update Velocity and Position

Input: X^t, V^t, w, c_1, c_2, r_1, r_2, $\hat{P}^t$,$\hat{G}^t$,;
Output: V^{t+1}, X^{t+1} ;
Set the 'block size' and 'grid size', tx is the threads number obtained by Eq. (1);
for $tx := 0$ *to* N *in parallel* **do**
  $V_{tx}^{t+1} := w \cdot V_{tx}^t + c_1 \cdot r_1 \cdot \left(\hat{P}_{tx}^t - X_{tx}\right) + c_2 \cdot r_2 \cdot \left(\hat{G}^t - X_{tx}\right)$;
  $X_{tx}^{t+1} := X_{tx}^t + V_{tx}^{t+1}$;
end
return V^{t+1}, X^{t+1};

Table 3
Benchmark test functions.

Number	Name	Equation	Feature	Bounds	Reference
F1	Sphere	$f_{(x)} = \sum_{i=1}^{D} x_i^2$	US	$[-100, 100]^D$	[33]
F2	De Jong	$f_{(x)} = \sum_{i=1}^{D} \lvert x_i \rvert^{(i+1)}$	UN	$[-100, 100]^D$	[33]
F3	Easom	$f_{(x)} = -(-1)^D(\prod_1^D cos^2(x_i))exp[-\sum_{i=1}^{D}(x_i - \pi)^2]$	UN	$[-1, 1]^D$	[33]
F4	Elliptic	$f_{(x)} = \sum_{i=1}^{D}(10^6)^{\frac{i-1}{D-1}} x_i^2$	US	$[-100, 100]^D$	[34]
F5	Rosenbrock	$f_{(x)} = \sum_{i=1}^{D}[100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2]$	UN	$[-30, 30]^D$	[35]
F6	Step	$f_{(x)} = \sum_{i=1}^{D} \lfloor x_i + 0.5 \rfloor^2$	US	$[-100, 100]^D$	[32]
F7		$f_{(x)} = \sum_{i=1}^{D} \left[sin(x_i) + sin(\frac{2x_i}{3}) \right] + 1.21598D$	MS	$[-100, 100]^D$	[36]
F8	Ackley	$f_{(x)} = 20 + \varrho - 20 \cdot \varrho^{-0.2\sqrt{\sum_{i=1}^{D}/D}} - \varrho^{\sum_{i=1}^{D} cos(2\pi \cdot x_i)/D}$	MN	$[-500, 500]^D$	[37]
F9	Rastrigin	$f_{(x)} = \sum_{i=1}^{D} \left[x_i^2 - 10cos(2\pi \cdot x_i) + 10 \right]$	MS	$[-100, 100]^D$	[38]
F10	Griewank	$f_{(x)} = \frac{1}{4000} \sum_{i=1}^{D} x_i^2 - \prod_{i=1}^{D} cos(x_i/\sqrt{i}) + 1$	MN	$[-600, 600]^D$	[39]

Table 4
Experiment platform.

Name	Model
CPU	Intel(R) Core(TM) i7-8750H CPU @ 2.20 GHz
GPU	NVIDIA Quadro P3200
Operating system	Windows 10 based on X64 processor
Development environment	Microsoft Visual Studio 2017, CUDA 10.2

Table 5
Parameter settings.

Algorithm	Parameter settings
CPU-PSO	$\omega = [0.4, 0.9]$, $c_1 = 2, c_2 = 2, r_1 = rand\,[0, 1]\,, r_2 = rand\,[0, 1]$
GPU-PSO	$\omega = [0.4, 0.9]$, $c_1 = 2, c_2 = 2, r_1 = rand\,[0, 1]\,, r_2 = rand\,[0, 1]$

The computing platform used in this paper is shown in Table 4. To avoid the influence of initial data, GPU-PSO and CPU-PSO use the same random number seed to initialize data on the CPU. The main parameters used by the algorithm in this paper in the velocity update strategy are shown in Table 5. The meanings of the parameters in Table 3 can be obtained by referring to the relevant references.

6.1. The relation between the optimal function value and the population size

Population size is an important parameter in particle swarm optimization algorithm, which will directly affect the search ability and convergence speed of the algorithm. In this paper, we use the control variable method to explore the influence of particle size on the PSO algorithm to solve the optimization problem. In order to make the experimental results objective, reasonable and reliable, the CPU-PSO and GPU-PSO algorithms in this paper were independently executed 20 times for each standard test function. The results of 20 experiments were counted, including average, maximum, and minimum values. Among them, the dimensions of all test functions are $M = 32$, and the maximum number of iterations is 2000.

By analyzing the basic properties of these test functions, it is found that they can be divided into four categories: single-peak independent functions, single-peak dependent function, multi-peak independent functions and multi-front dependent function. In this paper, we first test the effect of particle size on the optimal value of the function for 10 benchmark functions. Considering the length of the paper, a representative function is selected from each of the above categories to draw the function fitness value and the relationship between the number of particles, as shown in Fig. 6. In Fig. 6, the horizontal axis represents the logarithm base 2 of the number of particles, the vertical axis represents the fitness value of the function, blue represents the maximum value, green represents the average value, and purple represents the minimum value.

It can be clearly seen from Fig. 6 that when the population size increases from 2 to 2^{13}, the number of particles has two influences on the convergence rate. The first influence is shown in Figures (a) and (c). When the number of particles is in the interval $\left[2^1, 2^8\right]$, the convergence rate increases almost exponentially. When the number of particles exceeds 2^4 in Sphere or exceeds 2^8 in Ackley, the growth of the number of particles contributes little to the convergence rate. In contrast, the second effect is shown in figure (d). The convergence rate is almost linear where the number of particles is in the whole interval $\left[2^1, 2^{13}\right]$. But it found that the accuracy of the optimal value has no obvious improvement when the number of particles reaches.

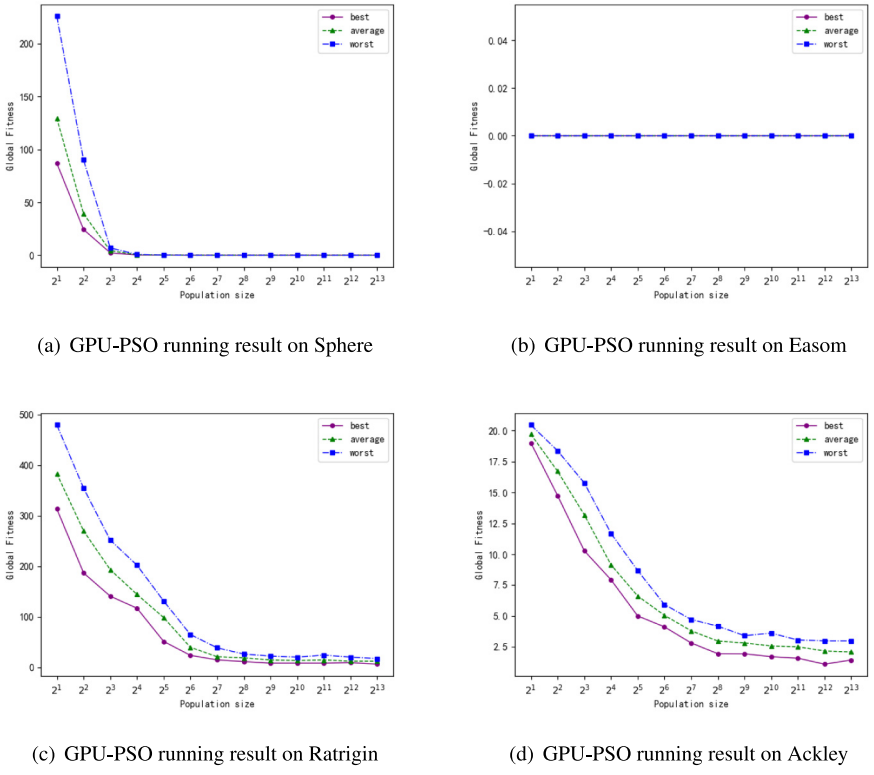
The reason is that when the population size is small, due to fewer individuals in the population, the search space is less explored, and it is easy to fall into the local optimal solution. With the increase of population size, the search space is more fully explored, the searchability is gradually enhanced, and the optimization effect is gradually improved. However, when the population size reaches a certain level, the improvement effect of increasing the population size on the optimization effect becomes very small, or even no significant improvement. This is because with the increase of population size, the amount of computation and storage of the algorithm will also increase, which will lead to the slow convergence speed of the algorithm, and even the phenomenon of over-fitting, thus affecting the optimization effect of the algorithm. To sum up, when using the GPU-PSO algorithm to solve single-objective optimization problems, a large population is also necessary.

6.2. The relation between dimension and speedup ratio

Speedup ratio is one of the most commonly used indicators for parallel computing performance evaluation. In order to avoid the interference caused by the first and last data transmission time between CPU and GPU, the speedup ratio used in this paper is "pure speedup ratio", that is, only the time consumed by flight iteration is calculated when the speedup ratio is calculated. The specific definition is as follows

$$S = \frac{T_g}{T_c} \tag{9}$$

Where, T_g represents the iteration time of the GPU-PSO algorithm under the same conditions (particle number, dimension, iteration



(a) GPU-PSO running result on Sphere

(b) GPU-PSO running result on Easom

(c) GPU-PSO running result on Ratrigin

(d) GPU-PSO running result on Ackley

Fig. 6. The relation between the optimal function value and the population size for four function on GPU-PSO.

Table 6
The result of Sphere function on GPU-PSO and CPU-PSO.

Dimension	Running result		Running time		Speedup
	CPU-PSO value	GPU-PSO value	CPU-PSO time	GPU-PSO time	
4	6.0665572E−139	0.0000000E+00	11.35394788	0.541856518	20.95
8	3.3768764E−126	0.0000000E+00	20.07299924	0.550002079	36.50
16	9.0308011E−98	0.0000000E+00	38.83305407	0.56899621	68.25
32	3.0580763E−07	1.1409081E−17	80.06103563	0.589381475	135.84
64	2.4335207E−04	2.1442233E−05	154.7250023	0.656248808	235.77
128	4.5685425E−03	2.0323400E−03	320.514792	0.671871424	477.05
256	8.8556498E−02	8.2393450E−02	665.4135034	1.05546999	630.44

Table 7
The result of De Jong function on GPU-PSO and CPU-PSO.

Dimension	Running result		Running time		Speedup
	CPU-PSO value	GPU-PSO value	CPU-PSO time	GPU-PSO time	
4	5.7991800E−62	0.0000000E+00	11.89298558	0.553203096	21.50
8	1.4621025E−47	7.7605030E−37	22.17996383	0.562011003	39.47
16	4.2571833E−37	9.0605650E−31	41.48499656	0.588454752	70.50
32	6.8602596E−34	1.2281144E−25	81.39999723	0.606755018	134.16
64	2.5768028E−27	3.2370202E−22	166.1309953	0.60685231	273.76
128	2.0829954E−25	4.1811630E−22	313.3009646	0.743712187	421.27
256	2.1358487E−26	1.4872195E−20	637.6970599	1.151456118	553.82

number, etc.), and the unit is second. T_c represents the iteration time of CPU-PSO algorithm under the same conditions, in a unit of second. S is the pure speedup ratio used in this paper.

Based on the above experiments, in order to further study the pure speedup ratio of the GPU-PSO algorithm and CPU-PSO algorithm, the population size of these two algorithms is set as 1024 and the maximum iteration number is 2000. They conducted 20 independent experiments on different dimensions of 10 standard test functions respectively, and the results of average value, running time and pure speedup ratio of the two are shown in Tables 6–15.

By analyzing the data in the tables and figures, we can draw the following conclusions.

(1) Optimization Performance Comparison: Compared with CPU-PSO algorithm, GPU-PSO algorithm achieves approximate or better solution accuracy in any dimension of other test functions except Rosenbrock function in dimension 2^6, 2^7, 2^8, and Ackley function in dimension 2^8. The reason is that the calculation accuracy of floating-point numbers is different between GPU and CPU. In summary, the GPU-PSO algorithm and CPU-PSO algorithm have the same significant convergence and stability.

(2) Running Time: As can be seen from Fig. 7, when the CPU-PSO algorithm is used to solve the test function, the increasing gradient of time has a linear relationship with the increasing gradient of dimension. Furthermore, we can see that the execution time of CPU-PSO is positively correlated with the complexity

Table 8
The result of Eesom function on GPU-PSO and CPU-PSO.

Dimension	Running result		Running time		Speedup
	CPU-PSO value	GPU-PSO value	CPU-PSO time	GPU-PSO time	
4	0.0000000E+00	0.0000000E+00	38.70700121	0.582329512	66.47
8	0.0000000E+00	0.0000000E+00	76.10396433	0.591484547	128.67
16	0.0000000E+00	0.0000000E+00	151.8719585	0.633552313	239.71
32	0.0000000E+00	0.0000000E+00	295.137079	0.668755903	441.32
64	0.0000000E+00	0.0000000E+00	601.7759991	0.710742474	846.69
128	0.0000000E+00	0.0000000E+00	1204.941035	0.970243454	1241.90
256	0.0000000E+00	0.0000000E+00	2447.633089	1.75876379	1391.68

Table 9
The result of Elliptic function on GPU-PSO and CPU-PSO.

Dimension	Running result		Running time		Speedup
	CPU-PSO value	GPU-PSO value	CPU-PSO time	GPU-PSO time	
4	1.3541400E−47	0.0000000E+00	14.12948108	0.556823263	25.38
8	3.1389658E−40	5.4692676E−36	24.2238009	0.576194983	42.04
16	1.4120644E−27	2.7242950E−28	45.43726444	0.598071356	75.97
32	1.6231583E−13	5.7342830E−19	89.55806184	0.640622377	139.80
64	6.2571651E−04	5.6419120E−07	191.0419173	0.609361887	313.51
128	2.3225867E+01	2.9308724E+00	398.9767647	0.765585899	521.14
256	8.3313671E+03	5.8340137E+03	756.7315586	1.156288385	654.45

Table 10
The result of Rosenbrock function on GPU-PSO and CPU-PSO.

Dimension	Running result		Running time		Speedup
	CPU-PSO value	GPU-PSO value	CPU-PSO time	GPU-PSO time	
4	0.0000000E+00	0.0000000E+00	20.73652411	0.593713999	34.93
8	0.0000000E+00	0.0000000E+00	44.51287603	0.594758106	74.84
16	1.0543167E−24	1.5667467E−12	92.31920981	0.609405518	151.49
32	4.7922847E+00	9.4384700E−03	186.856005	0.630572309	296.33
64	4.6567031E+01	5.0978695E+01	388.7349908	0.640585899	606.84
128	1.1807143E+02	1.2312118E+02	768.9551203	0.797002792	964.81
256	2.5175359E+02	2.5390993E+02	1530.274873	1.256081104	1218.29

Table 11
The result of Step function on GPU-PSO and CPU-PSO.

Dimension	Running result		Running time		Speedup
	CPU-PSO value	GPU-PSO value	CPU-PSO time	GPU-PSO time	
4	0.0000000E+00	0.0000000E+00	13.79266357	0.593713522	23.23
8	0.0000000E+00	0.0000000E+00	26.29311943	0.611896276	42.97
16	0.0000000E+00	0.0000000E+00	49.0174408	0.619391928	79.14
32	0.0000000E+00	0.0000000E+00	99.04992414	0.640657425	154.61
64	0.0000000E+00	1.0000000E+00	194.497854	0.687675953	282.83
128	1.1000000E+01	4.0000000E+00	399.2347436	0.921973467	433.02
256	3.6000000E+01	2.3000000E+01	835.4170079	1.624994993	514.10

Table 12
The result of F7 function on GPU-PSO and CPU-PSO.

Dimension	Running result		Running time		Speedup
	CPU-PSO value	GPU-PSO value	CPU-PSO time	GPU-PSO time	
4	−8.7003236E−06	−3.6879000E−04	11.03319001	0.633748331	17.41
8	−1.7400647E−05	−7.3854000E−04	19.66239667	0.639373331	30.75
16	1.1584982E−04	−1.4427500E−03	36.63779998	0.640662193	57.19
32	7.9431478E+00	7.9387383E+00	85.27729321	0.713749886	119.48
64	1.9332582E+01	1.6870083E+01	163.4649394	0.787501669	237.77
128	3.7860446E+01	3.1676844E+01	332.4356558	0.953124762	348.79
256	1.0548455E+02	1.0201234E+02	615.6976006	1.343747377	458.19

of the test function by observing the low-dimension case. As shown in Fig. 8. If other parameters are set unchanged, when the problem dimension is no smaller, the increasing gradient of computing time of the GPU-PSO algorithm is extremely low for all functions, and the total time is within one second. In general, the execution time of GPU-PSO increases linearly with the increase of dimension. At the same time, it can be seen that the running time of the GPU-PSO algorithm is non-linearly related to the complexity of the test function.

(3) speedup Ratio: Fig. 9 shows that the pure speedup ratio of the two algorithms on each test function increases with the increase of dimension, and the higher the function complexity, the greater the pure speedup ratio. Among them, when the dimension of the Ackley function is 256, the pure speedup ratio is more than 2000 times.

The number of cores on a GPU determines the number of threads it can process simultaneously. If the GPU has more cores, it can handle more threads simultaneously, which increases thread

Table 13
The result of Ackley function on GPU-PSO and CPU-PSO.

Dimension	Running result		Running time		Speedup
	CPU-PSO value	GPU-PSO value	CPU-PSO time	GPU-PSO time	
4	8.2548400E−08	8.2548404E−08	87.34434366	0.633820524	137.81
8	8.2548410E−08	8.2548404E−08	145.0533125	0.637239695	227.63
16	8.2548425E−08	8.2548420E−08	274.0419569	0.646325378	424.00
32	2.5012221E+00	1.1167961E+00	515.5443928	0.648406982	795.09
64	3.3045955E+00	3.0648250E+00	987.9207883	0.753452301	1311.19
128	4.3628760E+00	4.2185700E+00	1911.64064	0.933253765	2048.36
256	5.2820790E+00	5.4011097E+00	3788.150329	1.547958851	2447.19

Table 14
The result of Rastrigin function on GPU-PSO and CPU-PSO.

Dimension	Running result		Running time		Speedup
	CPU-PSO value	GPU-PSO value	CPU-PSO time	GPU-PSO time	
4	.0000000E+00	0.0000000E+00	14.53124785	0.599531164	24.24
8	9.9495906E−01	0.0000000E+00	28.3135879	0.602465668	47.00
16	6.9647134E+00	5.9698105E+00	54.37457585	0.610782118	89.02
32	1.5937542E+01	1.1948089E+01	113.4680221	0.625000954	181.55
64	3.9074832E+01	2.6875364E+01	236.9318812	0.646872854	366.27
128	1.7074401E+02	6.7981926E+01	472.6261415	0.812581539	581.64
256	3.94E+02	1.2858865E+02	944.1592801	1.390626192	678.95

Table 15
The result of Griewank function on GPU-PSO and CPU-PSO.

Dimension	Running result		Running time		Speedup
	CPU-PSO value	GPU-PSO value	CPU-PSO time	GPU-PSO time	
4	2.4743745E−02	2.4636940E−02	32.66735005	0.621034384	52.60
8	1.6518410E−01	6.3984950E−02	61.5668149	0.629741201	97.77
16	7.7057497E−02	1.1102230E−16	116.6156487	0.632370958	184.41
32	9.3222282E−02	7.3993800E−03	228.4820416	0.678780317	336.61
64	5.3976318E−01	7.4520800E−03	461.8195808	0.697093248	662.49
128	1.1026473E+00	2.1073010E−02	935.2199476	0.959671736	974.52
256	2.1748978E+00	2.2129470E−01	1916.542015	1.359134197	1410.12

concurrency. Similarly, if a GPU has fewer cores, it can only handle fewer threads simultaneously, thereby reducing thread concurrency. So, the number of GPU cores is one of the important factors affecting the concurrency of threads. Therefore, in the GPU-PSO algorithm, the number of GPU cores will directly affect the particle size and dimension size that can be processed in parallel, which determines the performance improvement effect of the algorithm. In addition, there is another important reason why the GPU-PSO algorithm proposed in this paper has better performance improvement on these 10 test functions. The reason is that the GPU-PSO algorithm decomposes its calculation steps when calculating the fitness value of the high-dimensional complex test function and adopts the calculation mode combining fine-grained parallel with coarse-grained parallel. This is one of the characteristics and shortcomings of the algorithm in this paper, indicating that when the parallel algorithm solves different optimization problems, the parallel mode designed according to the problem itself can maximize and reduce the time cost.

7. Conclusions and perspectives

This study proposes an efficient parallel implementation of the PSO algorithm on the GPU platform based on NVIDIA's CUDA architecture platform. Based on the characteristics of different memory hierarchies and different execution models of CUDA, we design specific data structures to improve the efficiency of data-parallel processing. Second, we ensure that each computational aspect of the algorithm is implemented on the GPU, and the whole process involves only two data transfers, which successfully reduces the communication time between CPU and GPU. In addition, we also utilize a pattern of merged memory accesses aimed at improving the efficiency of data access. Finally, the effectiveness of the algorithm was verified on 10 standard benchmark test functions. The experimental results show that the GPU-PSO algorithm obtains similar experimental results to the CPU-PSO algorithm and significantly reduces the execution time with a maximum pure speedup ratio of 2000 times on the high-dimensional Ackley function. The GPU-PSO algorithm occupies less than 100 percent of the threads only in the statute parallel mode used in the process of finding the globally optimal individual, and all other phases are 100 percent, which achieves efficient use of resources.

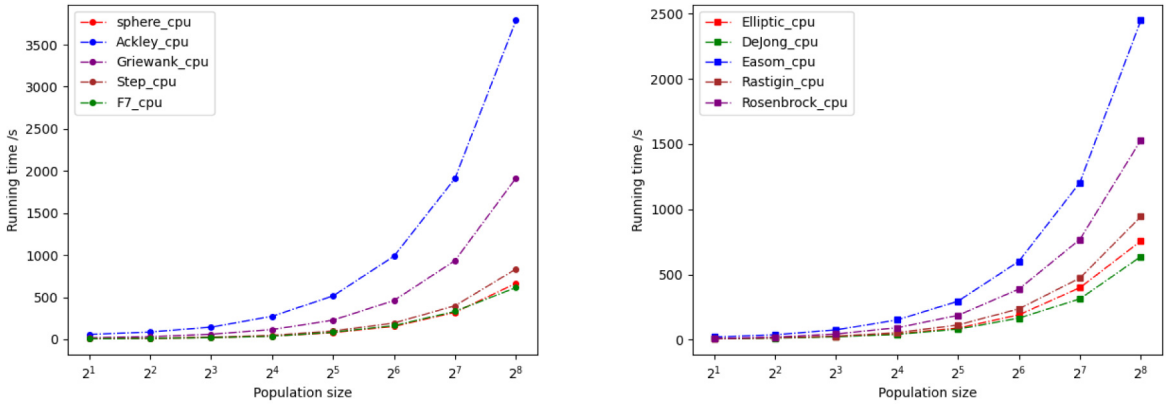
Future work includes investigating more complex parallel modes, as well as extending the tests to platforms with multiple GPU cards and evaluating the behavior of the mechanisms applied to control communication between different GPU cards. Designing multi-intelligence algorithms combining parallel computing and distribution to solve practical large-scale engineering optimization problems.

CRediT authorship contribution statement

Yanhong Zhuo: Methodology, Visualization, Software, Writing – original draft , Writing – review & editing. **Tao Zhang:** Conceptualization, Methodology. **Feng Du:** Data curation, Investigation. **Ruilin Liu:** Validation, Writing – review & editing.

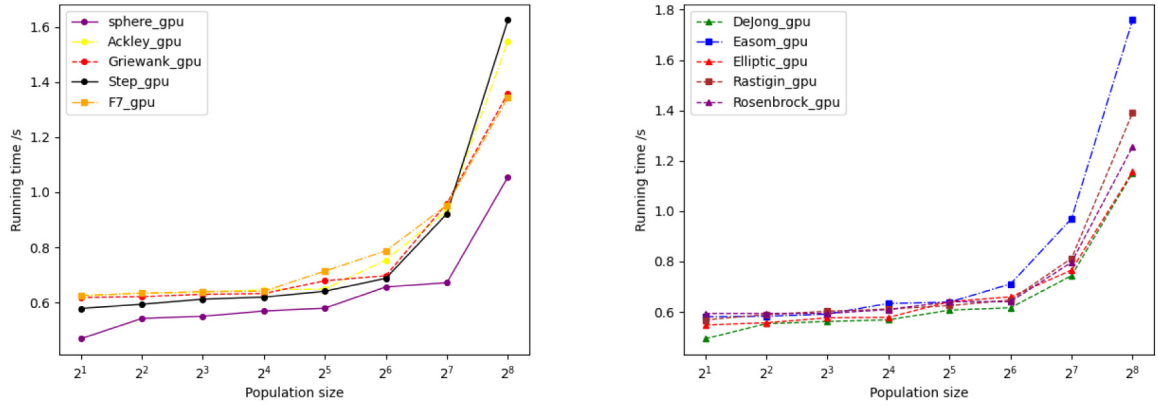
Declaration of competing interest

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.



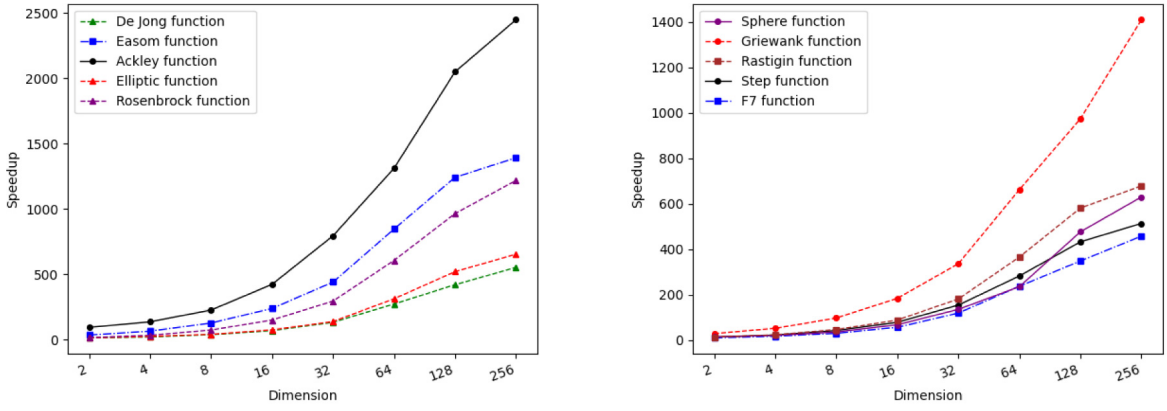
(a) CPU-PSO running time (b) CPU-PSO running time

Fig. 7. The running time of ten test function on CPU-PSO.



(a) GPU-PSO running time (b) GPU-PSO running time

Fig. 8. The running time of ten test function on GPU-PSO.



(a) (b)

Fig. 9. The speedup of ten test function about CPU-PSO and GPU-PSO.

Data availability

Data will be made available on request.

References

[1] K. Ihara, S. Kato, T. Nakaya, T. Ogi, H. Masuda, Application of pso-based constrained combinatorial optimization to segment assignment in shield tunneling, in: International Conference on Agents and Artificial Intelligence, Springer, 2019, pp. 166–182.

[2] B. Song, Z. Wang, L. Zou, An improved PSO algorithm for smooth path planning of mobile robots using continuous high-degree Bezier curve, Appl. Soft Comput. 100 (2021) 106960.

[3] K. Chen, F. Zhou, A. Liu, Chaotic dynamic weight particle swarm optimization for numerical function optimization, Knowl.-Based Syst. 139 (2018) 23–40.

[4] H. Zhou, M. Song, W. Pedrycz, A comparative study of improved GA and PSO in solving multiple traveling salesmen problem, Appl. Soft Comput. 64 (2018) 564–580.

[5] B.M. Sahoo, H.M. Pandey, T. Amgoth, GAPSO-H: A hybrid approach towards optimizing the cluster based routing in wireless sensor network, Swarm Evol. Comput. 60 (2021) 100772.

[6] W. Zhu, H.N. Rad, M. Hasanipanah, A chaos recurrent ANFIS optimized by PSO to predict ground vibration generated in rock blasting, Appl. Soft Comput. 108 (2021) 107434.

[7] S.U. Mane, P.S. Lokare, H.R. Gaikwad, Overview and applications of GPGPU based parallel ant colony optimization, 2022, arXiv preprint arXiv:2203.11487.

[8] A. Borisenko, S. Gorlatch, Efficient GPU-parallelization of batch plants design using metaheuristics with parameter tuning, J. Parallel Distrib. Comput. 154 (2021) 74–81.

[9] A.B. Miranda, D.J. Molinares, C.J.A. Hernandez, L.G. Reyes, J. Ruiz-Rangel, Adaptation of parallel framework to solve traveling salesman problem using genetic algorithms and tabu search, 2021.

[10] K. Huang, J. Cao, Parallel differential evolutionary particle filtering algorithm based on the CUDA unfolding cycle, Wirel. Commun. Mob. Comput. 2021 (2021).

[11] L.d.P. Veronese, R.A. Krohling, Swarm's flight: Accelerating the particles using C-CUDA, in: 2009 IEEE Congress on Evolutionary Computation, IEEE, 2009, pp. 3264–3270.

[12] Y. Zhou, Y. Tan, GPU-based parallel particle swarm optimization, in: 2009 IEEE Congress on Evolutionary Computation, IEEE, 2009 pp. 1493–1500.

[13] L. Mussi, Y.S. Nashed, S. Cagnoni, GPU-based asynchronous particle swarm optimization, in: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, 2011, pp. 1555–1562.

[14] Y. Hung, W. Wang, Accelerating parallel particle swarm optimization via GPU, Optim. Methods Softw. 27 (1) (2012) 33–51.

[15] R.M. Calazan, N. Nedjah, L. de Macedo Mourelle, Parallel GPU-based implementation of high dimension particle swarm optimizations, in: 2013 IEEE 4th Latin American Symposium on Circuits and Systems, LASCAS, IEEE, 2013, pp. 1–4.

[16] M.M. Hussain, H. Hattori, N. Fujimoto, A CUDA implementation of the standard particle swarm optimization, in: 2016 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC, IEEE, 2016, pp. 219–226.

[17] E.H. Silva, C.J. Bastos Filho, PSO efficient implementation on GPUs using low latency memory, IEEE Lat. Am. Trans. 13 (5) (2015) 1619–1624.

[18] S. Zhang, D. Royer, S.-T. Yau, GPU-assisted high-resolution, real-time 3-D shape measurement, Opt. Express 14 (20) (2006) 9120–9129.

[19] M.J. Flynn, Flynn's taxonomy, 2011.

[20] A. Ignatov, K. Byeoung-Su, R. Timofte, A. Pouget, Fast camera image denoising on mobile gpus with deep learning, mobile ai 2021 challenge: Report, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 2515–2524.

[21] M.K. Mukundan, R. Muthuganapathy, A parallel algorithm for computing voronoi diagram of a set of circles using touching disc and topology matching, Comput. Aided Geom. Design 94 (2022) 102079.

[22] T.R. Gadekallu, D.S. Rajput, M. Reddy, K. Lakshmanna, S. Bhattacharya, S. Singh, A. Jolfaei, M. Alazab, A novel PCA–whale optimization-based deep neural network model for classification of tomato plant diseases using GPU, J. Real-Time Image Process. 18 (4) (2021) 1383–1396.

[23] W. Han, H. Li, M. Gong, J. Li, Y. Liu, Z. Wang, Multi-swarm particle swarm optimization based on CUDA for sparse reconstruction, Swarm Evol. Comput. 75 (2022) 101153.

[24] K. Karimi, N.G. Dickson, F. Hamze, A performance comparison of CUDA and opencl, 2010, arXiv preprint arXiv:1005.2581.

[25] P. Pacheco, Parallel Programming with MPI, Morgan Kaufmann, 1997.

[26] J. Sanders, E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010.

[27] W. Liu, B. Vinter, A framework for general sparse matrix–matrix multiplication on GPUs and heterogeneous processors, J. Parallel Distrib. Comput. 85 (2015) 47–61.

[28] M. Harris, S. Sengupta, J.D. Owens, Parallel prefix sum (scan) with CUDA, GPU Gems 3 (39) (2007) 851–876.

[29] A. Ladikos, S. Benhimane, N. Navab, Efficient visual hull computation for real-time 3D reconstruction using CUDA, in: 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, IEEE, 2008, pp. 1–8.

[30] C. Tian, L. Fei, W. Zheng, Y. Xu, W. Zuo, C.-W. Lin, Deep learning on image denoising: An overview, Neural Netw. 131 (2020) 251–275.

[31] J. Kennedy, R. Eberhart, Particle swarm optimization, in: Proceedings of ICNN'95-International Conference on Neural Networks, Vol. 4, IEEE, 1995, pp. 1942–1948.

[32] 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.

[33] M. Molga, C. Smutnicki, Test Functions for Optimization Needs, Vol. 101, 2005, p. 48.

[34] P. Flajolet, J. Françon, Elliptic functions, continued fractions and doubled permutations (Ph.D. thesis), INRIA, 1987.

[35] Y.-W. Shang, Y.-H. Qiu, A note on the extended Rosenbrock function, Evol. Comput. 14 (1) (2006) 119–126.

[36] K. Krishnakumar, R. Swaminathan, S. Garg, S. Narayanaswamy, Solving large parameter optimization problems using genetic algorithms, in: Guidance, Navigation, and Control Conference, 1995, p. 3223.

[37] M.A. Potter, K.A.D. Jong, A cooperative coevolutionary approach to function optimization, in: International Conference on Parallel Problem Solving from Nature, Springer, 1994, pp. 249–257.

[38] H. Mühlenbein, M. Schomisch, J. Born, The parallel genetic algorithm as function optimizer, Parallel Comput. 17 (6–7) (1991) 619–632.

[39] A.O. Griewank, Generalized descent for global optimization, J. Optim. Theory Appl. 34 (1) (1981) 11–39.