Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

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

Parallel differential evolution with self-adapting control parameters and generalized opposition-based learning for solving high-dimensional optimization problems

Hui Wang[a,*], Shahryar Rahnamayan[b], Zhijian Wu[c]

[a] School of Information Engineering, Nanchang Institute of Technology, Nanchang 330099, PR China
[b] Faculty of Engineering and Applied Science, University of Ontario Institute of Technology (UOIT), 2000 Simcoe Street North, Oshawa, ON L1H 7K4, Canada
[c] State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, PR China

ARTICLE INFO

Article history:
Received 14 April 2011
Received in revised form
21 November 2011
Accepted 27 February 2012
Available online 4 March 2012

Keywords:
Differential evolution (DE)
Generalized opposition-based learning
Graphics processing units (GPU)
High-dimensional global optimization

ABSTRACT

Solving high-dimensional global optimization problems is a time-consuming task because of the high complexity of the problems. To reduce the computational time for high-dimensional problems, this paper presents a parallel differential evolution (DE) based on Graphics Processing Units (GPUs). The proposed approach is called GOjDE, which employs self-adapting control parameters and generalized opposition-based learning (GOBL). The adapting parameters strategy is helpful to avoid manually adjusting the control parameters, and GOBL is beneficial for improving the quality of candidate solutions. Simulation experiments are conducted on a set of recently proposed high-dimensional benchmark problems with dimensions of 100, 200, 500 and 1,000. Simulation results demonstrate that GjODE is better than, or at least comparable to, six other algorithms, and employing GPU can effectively reduce computational time. The obtained maximum speedup is up to 75.

1. Introduction

Many real world problems can be formulated as optimization problems. As their complexity increases, traditional optimization algorithms fail to converge at acceptable rates, if at all, and effective algorithms are required. An unconstrained minimization problem can be presented as follows:

$$\text{MIN } f(x)$$

where $x = [x_1, x_2, \dots, x_D]$ and D indicates the dimension of the problem.

In the past decades, different kinds of nature-inspired optimization algorithms have been designed and applied to solve challenging optimization problems, e.g., Simulated Annealing (SA) [21], Evolutionary Algorithms (EAs) [3], Differential Evolution (DE) [34], Particle Swarm Optimization (PSO) [20], Ant Colony Optimization (ACO) [9], Estimation of Distribution Algorithms (EDA) [22], etc. Although these algorithms have shown promising performance in solving lower dimensional problems ($D < 100$), many of them suffer from the curse of dimensionality [5], which implies that their performance deteriorates as the dimensionality of the search space increases.

Recently, some excellent evolutionary approaches have been proposed to solve high-dimensional global optimization problems accurately [7,15,43,50,53]. Unfortunately, they are very time-consuming because of the high complexity of the problems. For instance, it cost about 104 h with Java version to complete the whole experiments of CEC-2010 special session and competition on large-scale global optimization (it was tested in a single thread on an Intel Core 2 Quad CPU Q6600 with 2.40 GHz in Matlab for Linux) [35]. Moreover, most of the mentioned algorithms have a time complexity of $O(D^2)$. It implies that the running time increases dramatically with the problem's dimensions.

To reduce the computational time for high-dimensional problems, this paper presents a parallel DE algorithm based on GPU (GOjDE), which is an enhanced version of generalized opposition-based differential evolution (GODE) [43]. In GOjDE, a self-adapting parameter tuning strategy is utilized to maximize performance of the algorithm. Moreover, GOjDE is run on multiprocessors of GPU in a parallel manner. That is helpful to accelerate the algorithm.

The rest of the paper is organized as follows. Section 2 presents a short review of DE and GPU computing. Section 3 provides a review of related works on high-dimensional global optimization. The proposed GOjDE is described in Section 4. Section 5 explains implementation of GOjDE on GPU. In Section 6, the experimental results and analysis are provided. Finally, the work is concluded in Section 7.

* Corresponding author.
E-mail addresses: wanghui_cug@yahoo.com.cn (H. Wang), shahryar.rahnamayan@uoit.ca (S. Rahnamayan), zhijianwu@whu.edu.cn (Z. Wu).

0743-7315/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.02.019

2. Background review

2.1. Differential evolution

Differential Evolution (DE), proposed by Storn and Price [34], is an effective, robust, and simple global optimization algorithm. According to frequently reported experimental studies, DE has shown better performance than many other evolutionary algorithms (EAs) in terms of convergence speed and robustness over comprehensive benchmark functions and real-world problems [27,39,41].

There are several variants of DE [34]. According to suggestions in [16], the *rand/1/exp* strategy shows a better performance to solve large-scale problems. Yang et al. [50] pointed out that the used benchmark functions are not really large-scale and most of them can be easily optimized dimension by dimension. For the exponential crossover, even if we use a large *CR*, the mutated dimensions are still small.

Let us assume that $X_i(t)(i = 1, 2, \dots, N_p)$ is the ith individual in population $P(t)$, where N_p is the population size, t is the generation number, and $P(t)$ is the population in the tth generation. The main idea of DE is to generate trial vectors. Mutation and crossover are used to produce new trial vectors, and selection determines which of the vectors will be successfully selected for the next generation.

Mutation—For each vector $X_i(t)$ in Generation t, a mutant vector V is calculated by

$$V_i(t) = X_{i_1}(t) + F\left(X_{i_2}(t) - X_{i_3}(t)\right), \quad i \neq i_1 \neq i_2 \neq i_3, \tag{1}$$

where $i = 1, 2, \dots, N_p$ and i_1, i_2, and i_3 are mutually different random integer indices in $[1, N_p]$. The population size N_p should satisfy $N_p \geq 4$ because i, i_1, i_2, and i_3 are different. $F \in [0, 2]$ is a real number that controls the amplification of the difference vector $(X_{i_2}(t) - X_{i_3}(t))$.

Crossover—Like genetic algorithms, DE also employs a crossover operator to build trial vectors $(U_i(t) = \{U_{i,1}(t), U_{i,2}(t), \dots, U_{i,D}(t)\})$ by recombining of two different vectors.

$$U_{i,j}(t) = \begin{cases} V_{i,j}(t), & \text{if } \mathrm{rand}_j(0,1) \leq CR \vee j = l \\ X_{i,j}(t), & \text{otherwise} \end{cases}, \tag{2}$$

where $CR \in (0, 1)$ is the predefined crossover probability, and $\mathrm{rand}_j(0, 1)$ is a random number within $(0, 1)$ for the jth dimension, and $l \in \{1, 2, \dots, D\}$ is a random parameter index.

Selection—A greedy selection mechanism is used as follows:

$$X_i(t) = \begin{cases} U_i(t), & \text{if } f(U_i(t)) \leq f(X_i(t)) \\ X_i(t), & \text{otherwise} \end{cases}. \tag{3}$$

Without loss of generality, this paper only considers minimization problems. If, and only if, the trial vector $U_i(t)$ is better than $X_i(t)$, then $X_i(t)$ is set to $U_i(t)$; otherwise, the $X_i(t)$ remains unchanged.

2.2. Computing based on GPU

GPU computing refers to the computer programming paradigm of using a GPU to perform parallel computations using a SIMD (Single Instruction Multiple Data) parallelization paradigm. Compute Unified Device Architecture (CUDA) is a general purpose parallel computing architecture developed by NVIDIA in 2006 [26]. CUDA allows the programmer to efficiently program in a manner similar to C language but to run on NVIDIA graphics cards. In CUDA, parallelized programs are run on a kernel of code. The code on the kernel is run through several thousands of treads. Individual threads run all of the code on the kernel, but with different data [1].

CUDA programming model consists of four parts: Host and Device, Kernels, Thread Hierarchy, and Memory Hierarchy.

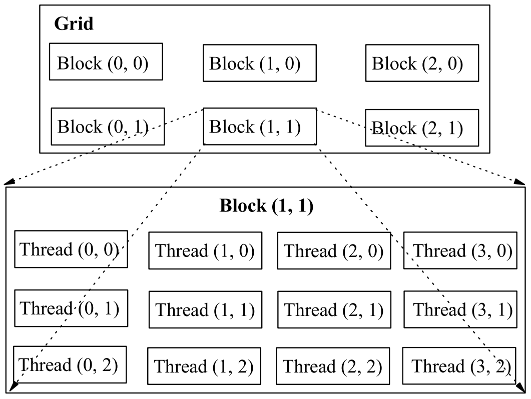


Fig. 1. The structure of grid, block and thread.

- Host and Device: CUDA's programming model assumes that CPU and GPU are regarded as a host and a coprocessor or device, respectively. In this model, CPU and GPU work collaboratively. CPU focuses on dealing with serial computation, and GPU concentrates on parallel computing.
- Kernels: CUDA allows programmers to define functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads.
- Thread Hierarchy: The kernels are organized as a grid. Each grid consists of several blocks, and each block contains several threads. Fig. 1 shows the structure of grid, block, and thread.
- Memory Hierarchy: There are four distinct memory, global memory, shared memory, local memory, and register.

The nVidia GeForce GTX 285 GPU hardware used in this paper has 30 multiprocessors and 240 cores. All threads are organized into blocks. Each block has a maximum of 512 threads, 16 kB shared memory and 16 kB registers. The total amount of constant memory is 64 kB. The threads in the same block can communicate through fast shared memory. Between the blocks, communication is possible only with slower global device memory [55].

Since the development of GPU, it has been applied in various areas for general-purpose computing. Wong [48] reported a parallel hybrid GA (HGA) on GPU. HGA extends the classical GA by incorporating the Cauchy mutation operator from evolutionary programming. In the parallel HGA, all steps except the random number generation procedure are performed in GPU. The obtained speedup increases with the growth of population size. The maximal speedup is up to 4.24 when the population size is set to 6400. Recently, Wong [47] implemented multi-objective evolutionary algorithms (MOEA) on GPU, where all steps except the non-dominated selection procedure are performed on GPU. Experimental results show that the speedups of parallel MOEA range from 5.62 to 10.75. It suggests that the proposed approach will be very useful for solving difficult multi-objective optimization problems which require huge population sizes.

Robilliard et al. [32] implemented Genetic Programming (GP) on GPU. The achieved speedup depends on the problem, notably on the presence of diverging operator and the number of fitness cases. The use of GPU fast memory can yield improvement up to factor 3 speedup. Banzhaf et al. [4] explained the general approach to use a GPU for GP and presented an overview of currently usable software systems. It pointed out that it would be helpful to identify categories of fitness functions that might and might not be suitable for implementation on GPUs. Zhou et al. [54] discussed the potential of GPU in high-dimensional optimization problems. Numerical examples compare the performance of CPU and GPU implementations of three classical minorization–maximization (MM) algorithm, nonnegative matrix factorization, PET image reconstruction, and multidimensional scaling. The attained speedup was up to 100.

Hu et al. [19] presented an analysis of parallel evolutionary algorithm with variable population size. The performance of the proposed algorithm is compared to conventional fixed-population-size genetic algorithms. The results show that variable population size can effectively improve performance.

Veronese and Krohling [40] proposed a DE algorithm based on GPU, in which only the fitness evaluation function is implemented in parallel, while the rest operations are sequential. Zhu [55] presented a novel DE (DE-PS) with local search and GPU which is run on single instruction-multiple thread (SIMD). The reported results indicate the GPU-accelerated SIMD-DE-PS method is orders of magnitude faster than the corresponding CPU implementation. In [55], simulation experiments are conducted on a set of function optimization problems with $D \leq 105$.

3. Overview of high-dimensional global optimization

In the past several years, the research on solving large-scale global optimization problems has attracted much attention. Some excellent works have been proposed. In this section, a brief overview of these approaches is presented.

Yang et al. [49] proposed a multilevel cooperative co-evolution algorithm based on self-adaptive neighborhood search DE (SaNSDE) to solve large scale problems. Hsieh et al. [18] presented an efficient population utilization strategy for PSO (EPUS-PSO) to manage the population size. Brest et al. [8] introduced a population size reduction mechanism into self-adaptive DE (jDEdynNP-F), where the population size decreases during the evolutionary process. Recently, Brest and Maučec proposed another version of jDEdynNP-F by employing three new mutation schemes [7].

Tseng and Chen [38] presented multiple trajectory search (MTS) by using multiple agents to search the solution space concurrently. Zhao et al. [52] used dynamic multi-swarm PSO with local search (DMS-PSO) for large scale problems. Rahnamayan and Wang [31] presented an experimental study of opposition-based DE (ODE) [30] on large scale problems. The reported results show that ODE significantly improves the performance of standard DE. Molina et al. [24] presented a memetic algorithm by employing MTS and local search chains to deal with large scale problems. Garcá-Martńez and Lozano [14] proposed a continuous variable neighborhood search algorithm based on evolutionary metaheuristic components. Muelas et al. [25] used a local search mechanism to improve the solutions obtained by DE. Duarte and Marti [10] presented an adaptive memory procedure based on scatter search and Tabu search to guide search in solving large scale problems. Wang et al. [43,44] used an enhanced ODE based on generalized opposition-based learning (GODE) to solve scalable benchmark functions.

Martinez et al. [15] introduced two mechanisms for improving the performance of DE on large-scale problems. The first one is role differentiation mechanism which defines the attributes for those vectors which are selected for each role. The second one is malleable mating which allows placing vectors to adapt their mating trends to ensure some similarity relations with the leading and correcting vectors. Yang et al. [50] proposed a generalized adaptive DE for large-scale optimization, which adopts the advantages of existing parameter adaptation schemes in DE. Zhao et al. [53] combined self-adaptive DE and multiple trajectory search (MTS) for large-scale optimization, which incorporates DE/current-to-*p*best [51] mutation strategy and hybridized with modified MTS.

Although the above mentioned algorithms achieve promising solutions in solving high-dimensional problems, they suffer from the same problem which the computational time significantly increases with the dimensions. In the current direction, this paper presents a new GOjDE algorithm based on GPU to reduce computational time using the parallelization.

4. DE with self-adapting control parameters and generalized opposition-based learning (GOjDE)

4.1. Generalized opposition-based learning

Opposition-based Learning (OBL) [37] is a new concept in computational intelligence, and has been proven to be an effective concept to enhance various optimization approaches [28–30,42]. When evaluating a solution x to a given problem, simultaneously computing its opposite solution will provide another chance for finding a candidate solution closer to the global optimum. Based on the concept of OBL, we propose a generalized OBL (GOBL) as follows [43–45]. Let x be a current solution, $x \in [a, b]$, and its opposite solution x^* is defined by:

$$x^* = k(a+b) - x \tag{4}$$

where k is a random number within [0, 1].

By staying within variables' interval static boundaries, we would jump outside of the already shrunken search space and the knowledge of the current converged search space would be lost. Hence, we calculate opposite particles by using dynamically updated interval boundaries $[a_j(t), b_j(t)]$ as follows [43].

$$X^*_{i,j} = k[a_j(t) + b_j(t)] - X_{i,j}, \tag{5}$$

$$a_j(t) = \min(X_{i,j}(t)), \qquad b_j(t) = \max(X_{i,j}(t)), \tag{6}$$

$$\begin{aligned} X^*_{i,j} \ = \ & \text{rand}(a_j(t), b_j(t)), \quad \text{If } X^*_{i,j}\langle X_{\min} \parallel X^*_{i,j}\rangle X_{\max}, \\ & i = 1, 2, \ldots, N_p, \quad j = 1, 2, \ldots, D, \ k = \text{rand}(0, 1), \end{aligned} \tag{7}$$

where $X_{i,j}$ is the jth element of vector of the ith candidate in the population, $X^*_{i,j}$ is the opposite candidate of $X_{i,j}$, $a_j(t)$ and $b_j(t)$ are the minimum and maximum values of the jth dimension in current search space, respectively, $\text{rand}(a_j(t), b_j(t))$ is a random number within $[a_j(t), b_j(t)]$, $[X_{\min}, X_{\max}]$ is the box-constraint of the problem, N_p is the population size, rand(0, 1) is a random number within [0, 1], k is generated anew in each generation (i.e. the same value of k is used for the whole population), and $t = 1, 2, \ldots,$ indicates the generations.

4.2. Self-adapting control parameters

The performance of DE algorithm highly depends on the control parameters F and CR. Different parameter settings will lead to different performances. To tackle this problem, some self-adaptive parameter strategies were proposed. Brest et al. [6] presented a novel self-adapting parameter mechanism, called jDE, which is simple yet effective. In jDE, each individual X_i has two independent control parameters F_i and CR_i, which are updated as follows.

$$F_i(t+1) = \begin{cases} F_l + \text{rand}_1 \cdot F_u, & \text{if rand}_2 < \tau_1 \\ F_i(t), & \text{otherwise} \end{cases}, \tag{8}$$

$$CR_i(t+1) = \begin{cases} \text{rand}_3, & \text{if rand}_4 < \tau_2 \\ CR_i(t), & \text{otherwise} \end{cases}, \tag{9}$$

where $\text{rand}_j, j \in \{1, 2, 3, 4\}$ are uniform random values $\in$ [0, 1]. τ_1 and τ_2 represent probabilities to adjust factors F and CR, respectively. In jDE, $\tau_1 = \tau_2 = 0.1$, $F_l = 0.1$ and $F_u = 0.9$. The new parameters F and CR take random values in [0.1, 1.0] and [0, 1], respectively.

The boundaries F_l and F_u is an empirical study in [6]. Their settings affect the performance of searching good solutions. In our experiments, the range of F is sensitive to solve large-scale problems. When changing the values of F_l and F_u, the performance of GODE is highly effected. In our studies, $F \in [0.2, 0.4]$ is a good setting. Therefore, we peak $F_l = F_u = 0.2$, and those values are used in this paper.

According to the studies of [23,34], the crossover probability $CR \in [0.8, 1.0]$ is a good choice. Therefore, we modify the CR updating model (Eq. (9)) as follows.

$$CR_i(t+1) = \begin{cases} CR_l + \text{rand}_3 \cdot CR_u, & \text{if rand}_4 < \tau_2 \\ CR_i(t), & \text{otherwise} \end{cases}, \tag{10}$$

where CR_l and CR_u are set to 0.8 and 0.2, respectively, in this paper. The new CR takes a value from $[0.8, 1.0]$ in a random manner.

4.3. The framework of GOjDE

Algorithm 1: The GOjDE Algorithm (CPU Implementation)

```
1  Initialization;
2  while FEs $\le$ MAX_FEs do
3      if $rand(0, 1) \le p_o$ then
           /* Conduct GOBL                                    */
4          Update the dynamic interval boundaries $[a_j(t), b_j(t)]$ in $P$
           according to (6);
5          $k = \text{rand}(0, 1)$;
6          for $i = 1$ to $N_p$ do
7              Generate the opposite individual $GOP_i$ of the $i$th
               individual $P_i$ according to (5);
8              Calculate the fitness value of $GOP_i$;
9              FEs ++;
10         end
11         Select $N_p$ fittest individuals from $\{P, GOP\}$ as new current
           population $P$;
12     end
13     else
           /* Execute the classical DE (rand/1/exp)  */
14         for $i = 1$ to $N_p$ do
15             Calculate the new parameters $F_i$ and $CR_i$ according to
               equations (8) and (10), respectively;
16             Generate the mutant vector $V_i$ according to (1);
17             Generate the trail vector $U_i$ according to (2);
18             Calculate the fitness value of $U_i$;
19             FEs ++;
20             Select a fitter one between $P_i$ and $U_i$ as the new $P_i$;
21         end
22     end
23 end
```

The framework of GOjDE on CPU is shown in Algorithm 1, where P is the current population, GOP is the opposite population after using GOBL, P_i is the ith individual in P, GOP_i is the opposite individual of P_i, k is a random number in $[0, 1]$, p_o is the probability of GOBL, $[a_j(t), b_j(t)]$ is the intervals of current population, FEs is the number of fitness evaluations, and MAX_FEs is the maximum number of evaluations. If the probability of GOBL is satisfied, then execute the GOBL strategy; otherwise execute DE. In DE, the control parameters F and CR are self-adjusting every generation.

5. Implementation of parallel GOjDE on GPU

In this paper, the parallel GOjDE is implemented on GPU featuring single instruction multiple threads (SIMT) execution (please see Chapter 4 in [26]). SIMT is used to manage hundreds of thread running several different programs. The multiprocessor maps each thread to one scalar processor core, and each scalar thread executes independently with its own instruction address and register state [26]. In GOjDE, the evolutionary operations on each individual are independent. Therefore, we can use a thread to execute the operations of an individual in a parallel manner. GOjDE consists of two parts: the original DE algorithm (including self-adapting control parameters) and generalized opposition-based jumping. For the first part, we use a kernel function DE_Kernel() to implement all operations of DE including the updating of the control parameters F and CR. The second part contains three operations: updating boundaries (see Eq. (6)), opposition (see Eq. (5)) and elite selection (see line 11 in Algorithm 1). We also use three kernel functions, Update_Boundaries_Kernel(), Opposition_Kernel() and Selection_Kernel(), to implement these operations, respectively.

5.1. DE_Kernel() function

The DE algorithm used in GOjDE contains five operations: parameter updating (see line 15 in Algorithm 1), mutation (see line 16 in Algorithm 1), crossover (see line 17 in Algorithm 1), fitness evaluation (see line 18 in Algorithm 1) and selection (see line 20 in Algorithm 1). These operations are independent for each individual. So, we allocate N_p threads to each individual and assure that each thread can execute all the operations of an individual.

5.2. Update_Boundaries_Kernel() function

The operation of updating boundaries (see line 11 in Algorithm 1) aims to find the minimum and maximum values of each dimension in the current search space. For each dimension, the operation of searching its boundaries is independent. Therefore, we can execute this operation in parallel. When the dimension is D, we need to allocate D threads to complete this task. In CUDA architecture, the maximum number of threads per block is 512. For $D = 1000$, we use two blocks to implement this kernel, and each block contains 500 threads.

5.3. Opposition_Kernel() function

The operation of opposition is independent for each individual. Therefore, we use N_p threads to generate all opposite individuals of current population in parallel. The fitness evaluation of each opposite individual is executed on its corresponding thread. This happens after generate opposite individuals.

5.4. Selection_Kernel() function

This kernel function differs from the former three operations, because the selection is not independent for each individual. The elite selection is to select N_p fittest individuals from $2 \times N_p$ individuals. To complete this task in parallel, we allocate a thread for each individual in current population P and opposite population GOP ($2 \times N_p$ individuals). For each individual, we compare it with other $2 \times N_p - 1$ individuals in $\{P \cup GOP\}$ and calculate its rank value. If the final rank value of an individual is less than N_p (it means that this individual is one of the N_p fittest individuals in $\{P \cup GOP\}$), then we select it as the new member entering to the next generation.

The main steps of GOjDE on GPU (GPU_GOjDE) are presented in Algorithm 2, where Iter is the number of iterations, and MAX_Iter is the maximum number of iterations. The relationship between MAX_FEs and MAX_Iter is MAX_Iter $= \frac{\text{MAX\_FEs}}{N_p}$.

6. Experimental studies

6.1. Experimental setup

There are 19 high-dimensional global optimization functions used for the following experiments. Functions $F_1 - F_6$ were chosen from the first six functions provided by CEC-2008 Special Session and Competition on Large Scale Global Optimization [36]. Functions $F_7 - F_{19}$ were proposed for the 2010 special issue of Soft Computing on scalability of evolutionary algorithms and other metaheuristics for large scale continuous optimization problems [17]. In this paper, we focus on investigating the optimization performance of GODE on problems with dimensions

Algorithm 2: The GOjDE Implemented on GPU (GPU_GOjDE)

```
1  Initialization;
2  while Iter ≤ MAX_Iter do
3      if $rand(0, 1) \le p_o$ then
           /* Conduct GOBL generation jumping */
4          Update_Boundaries_Kernel();
           /* The fitness evaluation of each opposite
              individual is conducted in
              Opposition_Kernel() */
5          Opposition_Kernel();
6          Selection_Kernel();
7      end
8      else
           /* Execute DE */
           /* The fitness evaluation of each individual
              is conducted in DE_Kernel() */
9          DE_Kernel();
10     end
11     Iter + +;
12 end
```

100, 200, 500 and 1000. The descriptions of these benchmark functions are listed in Table 1. For the specific definitions of these functions, please see [17].

The computational platform is as follows.

- System: Windows XP (SP2)
- CPU: Intel(R) Core(TM)2 Quad Q8200 (2.33 GHz)
- RAM: 2G
- GPU: NVIDIA GeForce GTX 285
- Compiler: Microsoft VS 2008.

To study the performance of GOjDE, three series of experiments are conducted as follows.

- GOjDE is compared with DE, GODE [43], CHC (Crossgenerational elitist selection, Heterogeneous recombination, and Cataclysmic mutation) [11] and G-CMA-ES (Restart Covariant Matrix Evolutionary Strategy) [2]. This experiment aims to investigate how the self-adapting parameter strategy and GOBL are helpful to improve the quality of solutions.
- The second experiment compares the computational time of GOjDE on CPU and GPU. Moreover, we also observe the obtained speedup. This experiment focuses on investigating whether GPU is helpful to reduce the computational time.
- The third experiment investigates the effects of population size on the speedup of GOjDE.

6.2. Comparison of GOjDE with DE, CHC, G-CMA-ES and GODE

In this section, we compare the performance of DE, CHC, G-CMA-ES, GODE and GOjDE on the test suite with $D = 100, 200$, 500 and 1000. This comparison aims to check whether the embedded strategies (self-adapting control parameter and GOBL) are beneficial for improving the quality of solutions.

To have a fair competition, the same settings are used for the common parameters. For DE, GODE and GOjDE, the population size (N_p) is fixed to 128. (This setting is different from [43].) The control parameters F and CR in DE and GODE are set to 0.5 and 0.9, respectively. For GOjDE, F and CR are self-adjusted during the evolution. The probability of GOBL p_o is set to 0.05. The parameter settings of CHC and G-CMA-ES are described in [16]. For all algorithms, the maximum number of fitness evaluations, MAX_FEs, is set to $5000 \times D$. Each run stops when the maximum number of evaluations (or MAX_Iter for GOjDE on GPU) is achieved. According to the suggestions of [17], all the results below 1E−14 have been approximated to 0.0.

Table 1
The 19 test functions used in the experiments, where $X \in R^n$ is the definition domain, and $f(x^o)$ is the minimum value of the function.

Functions	Name	X	$f(x^o)$
F_1	Shifted Sphere	$[-100, 100]$	−450
F_2	Shifted Schwefel 2.21	$[-100, 100]$	−450
F_3	Shifted Rosenbrock	$[-100, 100]$	390
F_4	Shifted Rastrigin	$[-5, 5]$	−330
F_5	Shifted Griewank	$[-600, 600]$	−180
F_6	Shifted Ackley	$[-32, 32]$	−140
F_7	Shifted Schwefel 2.22	$[-10, 10]$	0
F_8	Shifted Schwefel 1.2	$[-65.536, 65.536]$	0
F_9	Shifted Extended f_{10}	$[-100, 100]$	0
F_{10}	Shifted Bohacheysky	$[-15, 15]$	0
F_{11}	Shifted Schaffer	$[-100, 100]$	0
F_{12}	Hybrid shifted function	$[-100, 100]$	0
F_{13}	Hybrid shifted function	$[-100, 100]$	0
F_{14}	Hybrid shifted function	$[-5, 5]$	0
F_{15}	Hybrid shifted function	$[-10, 10]$	0
F_{16}	Hybrid shifted function	$[-100, 100]$	0
F_{17}	Hybrid shifted function	$[-100, 100]$	0
F_{18}	Hybrid shifted function	$[-5, 5]$	0
F_{19}	Hybrid shifted function	$[-10, 10]$	0

Tables 2–5 present the mean error values of the above five algorithms for $D = 100$, 200, 500 and 1000, respectively (The results of G-CMA-ES for $D = 1000$ are not included due to the large computation time for runs for some functions [17]). Results of CHC and G-CMA-ES are taken from [17]. The best results among the five (four for $D = 1000$) algorithms are shown in *bold*.

From the results, it can be seen that GOjDE achieves promising solutions on 11 functions, $F_1, F_5 - F_7, F_9 - F_{12}, F_{15}, F_{16}$ and F_{19}, while both DE and GODE obtain reasonable results on 9 functions, F_1, F_5-F_7, F_{10}, F_{12}, F_{15}, F_{16} and F_{19}. When the dimension increases to 1000, GOjDE, DE and GODE fail to solve F_7 and F_{15}. In our test, the fitness values of these two functions are larger than the maximum value (10^{308}) that double precision float number can represent. This problem can be solved by using higher precision data types, such as "long double" in C/C++. Although we used "long double" to represent the fitness value, GODE was implemented in Microsoft VS 2008, which uses the same size of bytes (8 bytes) to represent "double" and "long double". So, we did not list the results of these two functions for $D = 1000$.

GOjDE outperforms DE in all test cases except for F_3 and F_4. For F_3, DE performs better than GOjDE on $D = 100$, while GOjDE achieves better results on $D = 200$ and 500. For $D = 1000$, both DE and GOjDE obtain the same result. Compared to GODE, GOjDE performs better on all test functions except for F_4. On this function, GOjDE achieves better result than GODE on $D = 100$, while GODE outperforms GOjDE for the rest of dimensions. The comparison between GODE and GOjDE demonstrates that the achieved improvements of GOjDE are due to usage of the introduced self-adapting parameter strategy.

G-CMA-ES outperforms the other algorithms on three functions, F_2, F_3 and F_8. Especially for F_8, only G-CMA-ES achieves promising solution, while other algorithms fall into local minima. On function F_1, both G-CMA-ES and GOjDE could search the global optimum. For the rest the 15 function, GOjDE obtains better performance than G-CMA-ES. As pointed out in [43], CHC is not suitable for solving high-dimensional problems. It only obtains promising result on F_1, which is a simple and unimodal function. For the rest of the functions, the performance of CHC is seriously affected by the increasing of dimensions.

Fig. 2 shows the convergence curves of DE, GODE and GOjDE on four selected problems. As seen, GojDE converges faster than DE and GODE on the four problems. For F_1, DE shows faster convergence speed than GODE, while GODE converges faster than DE for the rest three problems.

Table 2
Mean error values for $D = 100$, where the best results are shown in *bold*.

Functions	DE Mean	CHC Mean	G-CMA-ES Mean	GODE Mean	GOjDE Mean
F_1	1.99E−11	3.56E−11	**0.00E+00**	1.04E−10	**0.00E+00**
F_2	1.60E+01	8.58E+01	**1.51E−10**	1.79E+01	1.42E+01
F_3	8.78E+01	4.19E+06	**3.88E+00**	8.83E+01	1.42E+02
F_4	**6.28E+01**	2.19E+02	2.50E+02	7.82E+01	6.70E+01
F_5	1.22E−11	3.83E−03	1.58E−03	6.36E−12	**0.00E+00**
F_6	8.81E−07	4.10E−07	2.12E+01	2.51E−06	**0.00E+00**
F_7	1.82E−06	1.40E−02	4.22E−04	4.67E−06	**3.25E−10**
F_8	4.54E+03	1.69E+03	**0.00E+00**	4.48E+03	7.92E+00
F_9	1.38E+00	5.86E+02	1.02E+02	1.15E+00	**0.00E+00**
F_{10}	6.96E−11	3.30E+01	1.66E+01	5.83E−11	**0.00E+00**
F_{11}	1.31E+00	7.32E+01	1.64E+02	1.29E+00	**0.00E+00**
F_{12}	6.80E−03	1.03E+01	4.17E+02	1.15E−03	**3.32E−05**
F_{13}	6.87E+01	2.70E+06	4.21E+02	6.87E+01	**6.59E+01**
F_{14}	1.92E+01	1.66E+02	2.55E+02	1.85E+01	**1.79E+01**
F_{15}	7.95E−07	8.13E+00	6.30E−01	2.29E−10	**0.00E+00**
F_{16}	3.46E−03	2.23E+01	8.59E+02	5.22E−04	**1.69E−04**
F_{17}	2.35E+01	1.47E+05	1.51E+03	2.18E+01	**1.94E+01**
F_{18}	4.99E−01	7.00E+01	3.07E+02	1.09E−01	**5.38E−02**
F_{19}	1.38E−08	5.45E+02	2.02E+01	4.34E−09	**0.00E+00**

Table 3
Mean error values for $D = 200$, where the best results are shown in *bold*.

Functions	DE Mean	CHC Mean	G-CMA-ES Mean	GODE Mean	GOjDE Mean
F_1	3.74E−11	8.34E−01	**0.00E+00**	3.22E−10	**0.00E+00**
F_2	4.19E+01	1.03E+02	**1.16E−09**	2.05E+01	1.20E+01
F_3	1.87E+02	2.01E+07	**8.91E+0**1	1.88E+02	1.84E+02
F_4	**1.41E+02**	5.40E+02	6.48E+02	1.58E+02	1.75E+02
F_5	1.42E−11	8.76E−03	0.00E+00	9.61E−12	**0.00E+00**
F_6	1.07E−06	1.23E+00	2.14E+01	2.61E−06	**0.00E+00**
F_7	3.66E−06	2.59E−01	1.17E−01	9.51E−06	**5.28E−09**
F_8	3.36E+04	9.38E+03	**0.00E+00**	3.02E+04	4.13E+02
F_9	2.82E+00	1.19E+03	3.75E+02	3.55E+00	**0.00E+00**
F_{10}	1.42E−10	7.13E+01	4.43E+01	8.84E−10	**0.00E+00**
F_{11}	2.78E+00	3.85E+02	8.03E+02	3.54E+00	**0.00E+00**
F_{12}	2.45E−02	7.44E+01	9.06E+02	3.61E−03	**8.70E−05**
F_{13}	1.43E+02	5.75E+06	9.43E+02	1.43E+02	**1.41E+02**
F_{14}	8.23E+01	4.29E+02	6.09E+02	7.87E+01	**6.43E+01**
F_{15}	1.94E−06	2.14E+01	1.75E+00	4.80E−09	**0.00E+00**
F_{16}	7.37E−03	1.60E+02	1.92E+03	1.27E−03	**3.26E−04**
F_{17}	4.61E+01	1.75E+05	3.36E+03	4.53E+01	**4.43E+01**
F_{18}	1.16E+00	2.12E+02	6.89E+02	6.32E−01	**8.21E−02**
F_{19}	6.53E−08	2.06E+03	7.52E+02	2.20E−08	**0.00E+00**

Table 4
Mean error values for $D = 500$, where the best results are shown in *bold*.

Functions	DE Mean	CHC Mean	G-CMA-ES Mean	GODE Mean	GOjDE Mean
F_1	1.13E−10	2.84E−12	**0.00E+00**	6.66E−10	**0.00E+00**
F_2	8.43E+01	1.29E+02	**3.48E−04**	4.95E+01	3.46E+01
F_3	4.83E+02	1.14E+06	**3.58E+02**	4.83E+02	4.78E+02
F_4	**3.99E+02**	1.91E+03	2.10E+03	4.67E+02	4.68E+02
F_5	1.95E−11	6.98E−03	2.96E−04	1.28E−11	**0.00E+00**
F_6	1.08E−06	5.16E+00	2.15E+01	2.92E−07	**0.00E+00**
F_7	9.76E−06	1.27E−01	7.21E+153	2.31E−05	**2.96E−08**
F_8	1.78E+05	7.22E+04	**2.36E−06**	1.52E+05	2.00E+04
F_9	7.16E+00	3.00E+03	1.74E+03	3.26E+00	**0.00E+00**
F_{10}	4.43E−10	1.86E+02	1.27E+02	2.41E−10	**0.00E+00**
F_{11}	7.32E+00	1.81E+03	4.16E+03	3.52E+00	**0.00E+00**
F_{12}	8.45E−02	4.48E+02	2.58E+03	6.26E−03	**3.12E−04**
F_{13}	3.66E+02	3.22E+07	2.87E+03	3.67E+02	**3.65E+02**
F_{14}	2.80E+02	1.46E+03	1.95E+03	2.37E+02	**2.10E+02**
F_{15}	4.52E−06	6.01E+01	2.82E+262	1.41E−08	**7.26E−10**
F_{16}	1.92E−01	9.55E+02	5.45E+03	3.21E−02	**7.69E−04**
F_{17}	1.23E+02	8.40E+05	9.59E+03	1.22E+02	**1.19E+02**
F_{18}	3.49E+00	7.32E+02	2.05E+03	1.24E+00	**3.83E−01**
F_{19}	2.64E−07	1.76E+03	2.44E+06	5.91E−07	**0.00E+00**

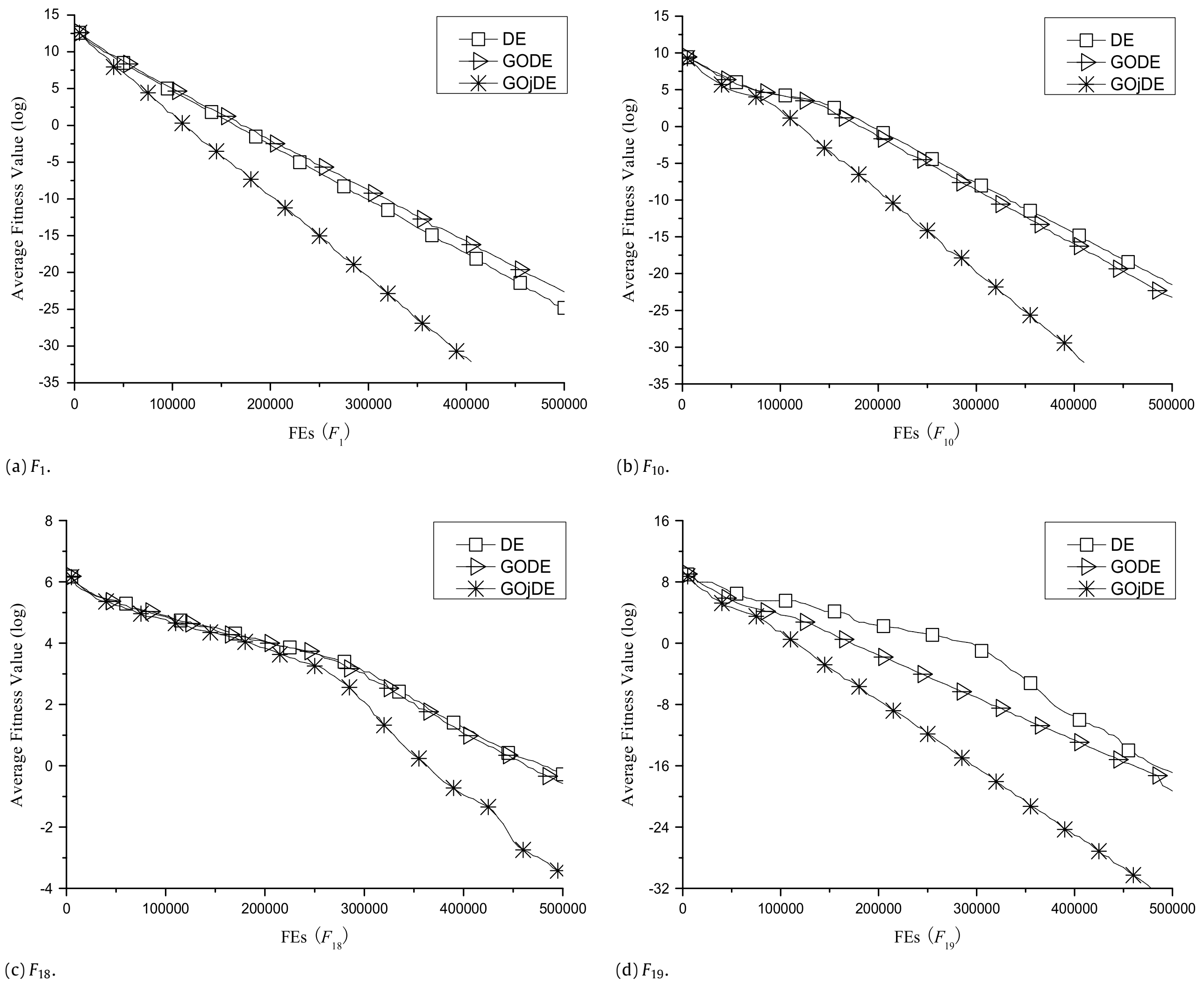
Table 5
Mean error values for $D = 1000$, where the best results are shown in *bold*.

Functions	DE Mean	CHC Mean	GODE Mean	GOjDE Mean
F_1	2.51E−10	1.36E−11	1.53E−09	**0.00E+00**
F_2	1.11E+02	1.44E+02	8.77E+01	**7.06E+01**
F_3	**9.78E+02**	8.75E+03	**9.78E+02**	**9.78E+02**
F_4	8.08E+02	4.76E+03	9.37E+02	**9.40E+02**
F_5	2.49E−11	7.02E−03	1.34E−11	**0.00E+00**
F_6	1.18E−06	1.38E+01	6.72E−07	**0.00E+00**
F_7	INF	3.52E−01	INF	INF
F_8	5.93E+05	3.11E+05	4.41E+05	**3.71E+05**
F_9	1.45E+01	6.11E+03	1.24E+01	**0.00E+00**
F_{10}	5.01E−09	3.83E+02	2.65E−09	**0.00E+00**
F_{11}	1.46E+01	4.82E+03	1.15E+01	**0.00E+00**
F_{12}	1.82E−01	1.05E+03	3.06E−02	**7.16E−04**
F_{13}	7.38E+02	6.66E+07	7.38E+02	**7.31E+02**
F_{14}	4.09E+02	3.62E+03	3.86E+02	**3.61E+02**
F_{15}	INF	8.37E+01	INF	INF
F_{16}	3.75E−01	2.32E+03	1.21E−01	**1.40E−03**
F_{17}	2.51E+02	2.04E+07	2.49E+02	**2.42E+02**
F_{18}	7.49E+00	1.72E+03	3.52E+00	**1.37E+00**
F_{19}	8.57E−07	4.20E+03	6.49E−07	**0.00E+00**

Non-parametric tests can be used for comparing algorithms whose results represent mean values for each problem, in spite of the inexistence of relationships among them. It is encouraged to use non-parametric tests to analyze the results obtained by evolutionary algorithms for continuous optimization problems in multiple problem analysis [12,13]. In this section, we conduct Friedman and Wilcoxon test to compare the performance of multiple algorithms on the test suite.

Table 6 presents the average rankings of GOjDE, GODE, DE, CHC, and G-CMA-ES for $D = 100$, 200, 500 and 1000, respectively. The computation of the results is used by the software MULTIPLETEST package (provided on the website: http://sci2s.ugr.es/sicidm). For each dimension, the performance of the five algorithms (four for $D = 1000$) can be sorted by the average rankings into the following order: GOjDE, GODE, DE, G-CMA-ES, and CHC. It means that GOjDE and CHC are the best and worst ones among the five algorithms, respectively.

Table 7 shows the p-values of applying Wilcoxon's test between GOjDE and other four algorithms for $D = 100$, 200, 500 and 1000. The computation of the results is used by SPSS Statistics software. In this paper, we only check the significance at a level of 0.05. The p-values below 0.05 are shown in **bold**. From the results, it can be seen that GOjDE is significantly better than GODE, DE, CHC and G-CMA-ES on each dimension.



(a) F_1. (b) F_{10}. (c) F_{18}. (d) F_{19}.

Fig. 2. The evolutionary processes of DE, GODE and GOjDE on four selected problems.

Table 6
Average rankings achieved by Friedman's test.

Algorithms	$D = 100$	$D = 200$	$D = 500$	$D = 1000$
GOjDE	4.66	4.68	4.71	3.76
GODE	3.34	3.29	3.45	2.79
DE	3.13	3.18	3.02	2.15
G-CMA-ES	2.24	2.37	2.08	N/A
CHC	1.63	1.47	1.74	1.29

Table 7
Wilcoxon's test between GOjDE and other algorithms, where the p-values below are shown in *bold*.

GOjDE vs.	$D = 100$	$D = 200$	$D = 500$	$D = 1000$
DE	**1.58E−02**	**1.94E−03**	**1.70E−03**	**6.13E−03**
CHC	**1.32E−04**	**1.32E−04**	**1.32E−04**	**3.60E−03**
G-CMA-ES	**3.78E−03**	**4.85E−03**	**1.74E−02**	N/A
GODE	**1.94E−03**	**1.94E−03**	**7.24E−04**	**2.28E−03**

Table 8
Comparison of GOjDE with GaDE and SOUPDE for $D = 200$, where the best results are shown in *bold*.

Functions	GaDE Mean	SOUPDE Mean	GOjDE Mean
F_1	**0.00E+00**	**0.00E+00**	**0.00E+00**
F_2	5.76E+01	2.33E+01	**1.33E+01**
F_3	**1.61E+01**	1.71E+02	1.89E+02
F_4	**0.00E+00**	2.27E−13	9.95E−01
F_5	**0.00E+00**	**0.00E+00**	**0.00E+00**
F_6	**0.00E+00**	6.44E−14	1.04E−14
F_7	**0.00E+00**	7.46E−14	**0.00E+00**
F_8	**3.02E+00**	2.46E+03	7.70E+02
F_9	4.53E−09	1.51E−05	**0.00E+00**
F_{10}	4.20E−02	**0.00E+00**	**0.00E+00**
F_{11}	1.85E−07	1.43E−05	**1.74E−08**
F_{12}	4.92E−14	**0.00E+00**	3.03E−04
F_{13}	1.24E+02	1.32E+02	**1.16E+02**
F_{14}	2.87E−12	**2.27E−13**	1.42E−12
F_{15}	**0.00E+00**	5.79E−14	**0.00E+00**
F_{16}	1.58E−12	**0.00E+00**	1.71E−13
F_{17}	**2.45E+01**	3.30E+01	4.91E+01
F_{18}	2.53E−08	**0.00E+00**	3.95E−12
F_{19}	**0.00E+00**	1.91E−14	**0.00E+00**
$w/t/l$	8/5/6	9/3/7	–

Table 9
Comparison of GOjDE with GaDE and SOUPDE for $D = 1000$, where the best results are shown in *bold*.

Functions	GaDE Mean	SOUPDE Mean	GOjDE Mean
F_1	**0.00E+00**	**0.00E+00**	**0.00E+00**
F_2	8.93E+01	9.25E+01	**6.59E+01**
F_3	9.45E+02	9.62E+02	**9.36E+02**
F_4	**0.00E+00**	2.00E−11	7.34E−03
F_5	**0.00E+00**	**0.00E+00**	**0.00E+00**
F_6	1.66E−14	3.42E−13	**1.53E−14**
F_7	**0.00E+00**	3.57E−13	INF
F_8	**1.77E+04**	2.12E+05	3.99E+05
F_9	**0.00E+00**	7.39E−05	4.56E−06
F_{10}	4.62E−01	**0.00E+00**	**0.00E+00**
F_{11}	**0.00E+00**	7.44E−05	3.95E−06
F_{12}	3.85E−12	**0.00E+00**	2.73E−04
F_{13}	7.15E+02	7.26E+02	**7.11E+02**
F_{14}	8.82E−11	**6.37E−12**	4.23E−11
F_{15}	**0.00E+00**	2.69E−13	INF
F_{16}	2.35E−12	**0.00E+00**	1.91E−12
F_{17}	**2.19E+02**	2.31E+02	2.57E+02
F_{18}	1.30E−07	**1.36E−12**	1.24E−07
F_{19}	3.78E−01	9.50E−14	**0.00E+00**
$w/t/l$	9/2/6	7/3/7	–

6.3. Comparison of GOjDE with other DE variants

To further verify the performance of the proposed approach, we compare GOjDE with two other DE variants published in the special issue of Soft Computing high-dimensional continuous optimization problems. The involved two DE algorithms include Shuffle Or Update Parallel Differential Evolution (SOUPDE) [46] and Generalized Adaptive Differential Evolution (GaDE) [50].

For the common parameters (such as N_p and MAX_FEs), the same values are used for GOjDE, GaDE and SOUPDE by the suggestions of [17]. For all the three algorithms, $N_p = 60$ and MAX_FEs $= 5000 \cdot D$. For GOjDE, The probability of GOBL p_o is set to 0.05. For the other parameters of GaDE and SOUPDE, please refer [46,50].

The mean error values achieved by GaDE, SOUPDE and GOjDE are presented in Tables 8 and 9. The comparison results between GOjDE and other algorithms are summarized as "$w/t/l$" in the last row of the table, which means that GOjDE wins in w functions, ties in t functions and loses in l functions, compared with its competitors. Results of GaDE are taken from Table 1 in [50]. Results of SOUPDE are taken from Tables 15 and 17 in [46]. In this paper, we only present the comparison results for $D = 200$ and $D = 1000$. For $D = 100$ and $D = 500$, we can get similar conclusions.

For $D = 200$, GOjDE outperforms GaDE on 8 functions, while GaDE achieves better results than GojDE on 6 functions. For the rest 5 functions, both GOjDE and GaDE can find the global optimum. SOUPDE performs better than GOjDE on 7 functions, but it achieves worse results on 9 functions. For the rest 3 functions, both of them obtain the same results. For $D = 1000$, GaDE achieves better results than GOjDE on 6 functions, while GOjDE outperforms GaDE on 9 functions. They achieve the same results on F_1 and F_5. GOjDE performs better than SOUPDE on 7 functions, while SOUPDE outperforms GOjDE on 7 functions, too. Both of them obtain the same performance on F_1, F_5 and F_{10}.

6.4. Comparison of the computational time of GOjDE on CPU and GPU

In this section, we investigate whether GPU can reduce the computational time of GOjDE. For the parameters of GOjDE, the same settings are used as described in Section 6.2. For GOjDE on GPU, N_p threads are allocated to each individual. Each run stops when the maximum number of generations (MAX_Iter $= \frac{5000 \times D}{N_p}$) is achieved.

Tables 10 and 11 present the computational time and speedup achieved by GOjDE. As seen, the computational time of GOjDE on CPU increases dramatically with increasing of dimensions, while GOjDE on GPU cost small. It demonstrates that GPU can reduce the computational time of GOjDE effectively. The obtained speedup is between 1.15 and 7.84. With the growth of dimensions, the speedup decreases in most cases. The main reason is that the number of threads is fixed to 128. When the dimension increases, the number of parallel individuals in population is fixed. When the dimension increases to 1000, the obtained 1.15 (for F_1) is not satisfactory. Under this case, the GPU can only save less computational time. To tackle this problem, we can increase the number of parallel individuals. The discussion about the effects of population size on the speedup is covered in Section 6.4.

6.5. Effects of population size on the speedup

In this section, we investigate the effects of population size on the speedup of GOjDE. In the implementation of GOjDE on GPU, the operations of each individual are executed by a thread. N_p individuals will need to allocate N_p threads. Larger population size will allocate more threads. In general, more threads can obtain

Table 10
Comparison of the computational time (in seconds) of GOjDE on CPU and GPU for $D = 100$ and $D = 200$.

Functions	$D = 100$			$D = 200$		
	CPU time	GPU time	Speedup	CPU time	GPU time	Speedup
F_1	2.83	1.36	2.08	7.64	4.64	1.65
F_2	4.11	1.34	3.07	12.56	4.50	2.79
F_3	17.25	2.20	7.84	64.20	8.67	7.40
F_4	9.02	1.54	5.86	31.61	5.63	5.61
F_5	10.66	2.00	5.33	38.81	7.14	5.44
F_6	8.80	2.36	3.73	31.13	8.41	3.70
F_7	4.36	1.43	3.04	13.36	4.51	2.96
F_8	2.84	1.38	2.06	7.58	4.61	1.64
F_9	15.69	3.31	4.74	58.55	14.41	4.06
F_{10}	7.50	2.94	2.55	26.30	10.83	2.43
F_{11}	13.16	3.78	3.48	49.28	14.23	3.46
F_{12}	6.48	1.84	3.52	21.83	6.36	3.43
F_{13}	15.11	2.70	5.60	56.00	14.69	3.81
F_{14}	11.02	1.98	5.57	39.70	6.86	5.79
F_{15}	5.77	2.14	2.70	19.63	7.69	2.55
F_{16}	9.59	1.89	5.07	34.20	6.53	5.24
F_{17}	15.66	2.31	6.78	58.53	9.14	6.40
F_{18}	14.36	2.08	6.90	53.22	10.01	5.32
F_{19}	7.47	2.16	3.46	29.00	12.58	2.31

Table 11
Comparison of the computational time (in seconds) of GOjDE on CPU and GPU for $D = 500$ and $D = 1000$.

Functions	$D = 500$			$D = 1000$		
	CPU time	GPU time	Speedup	CPU time	GPU time	Speedup
F_1	32.59	25.01	1.30	110.72	96.47	1.15
F_2	62.16	24.13	2.58	227.08	91.81	2.47
F_3	389.02	54.56	7.13	1532.97	223.95	6.84
F_4	181.55	32.59	5.57	709.33	129.20	5.49
F_5	227.36	40.25	5.65	892.39	159.81	5.58
F_6	178.42	47.84	3.73	695.47	190.50	3.65
F_7	65.34	22.68	2.88	–	–	–
F_8	31.11	24.08	1.29	105.70	91.11	1.16
F_9	350.83	85.23	4.12	1384.30	340.69	4.06
F_{10}	149.02	63.55	2.34	574.25	255.66	2.25
F_{11}	293.45	84.45	3.47	1154.19	337.75	3.42
F_{12}	120.34	47.30	2.54	463.74	171.25	2.71
F_{13}	337.09	74.03	4.55	1328.56	281.08	4.73
F_{14}	232.83	50.52	4.61	911.72	185.39	4.92
F_{15}	100.97	50.10	1.98	–	–	–
F_{16}	198.38	48.63	4.08	774.66	178.11	4.35
F_{17}	352.34	65.13	5.41	1389.61	245.23	5.67
F_{18}	316.22	53.36	5.93	1247.25	196.74	6.34
F_{19}	146.78	61.09	2.40	638.67	288.96	2.21

higher speedup. In this experiment, we set the population size N_p as 256, 1024, 2048 and 4096, respectively. For each problem with different N_p, the maximum number of generations (MAX_Iter) is set to 20,000. Each algorithm stops run when the maximum number of generations is achieved. For other parameters, we use the same settings as described in Section 6.2. Here, we only presents the results of problems on $D = 200$. For other dimensions, we can get similar conclusions.

Table 12 presents the computational results of GOjDE on CPU and GPU. When N_p varies from 256 to 4196, the computational effort increases about 16 times because we fix the maximum number of generations in the experiment. The computational time of GOjDE on CPU increases about 16 times, while GOjDE on GPU only increases $2.8 \sim 7.45$ times. With increasing of N_p, more threads are used to execute the operations of GPU_GODE in parallel and larger speedups are obtained. The maximum speedup is up to 75. It shows that GPU can effectively reduce the computational time when utilizing appropriate population size. Although larger population size could obtain higher speedup, the growth of speedup is determined by the parallel ability of the graphics card. The maximum number of the threads allocated to individuals is limited. When the allocated threads reach to the maximum limitation, the speedup will be up to the maximum value.

Storn and Price in [33] have indicated that a reasonable value for N_p could be chosen between $5 \cdot D$ and $10 \cdot D$. To investigate the effects of N_p for the quality of solutions, we compare the mean error values achieved by GojDE on GPU with different N_p. The comparison results are listed in Table 13. It can be seen that there is no fixed N_p for all test functions. For F_2 and F_8, $N_p = 1024$ is the best choice, while $N_p = 4096$ is the best for F_3, F_{13}. For the rest 14 functions, the best choice of N_p could be chosen between 1024 and 4096.

7. Conclusion

In this paper, we present a new parallel DE algorithm (GOjDE) based on GPU to solve high-dimensional global optimization problems. The proposed approach employs self-adapting control parameters and GOBL strategy to improve the quality of candidate solutions. Simulation experiments are conducted on 19 recently proposed high-dimensional benchmark problems with $D = 100$,

Table 12
Comparison of the computational time (in seconds) and speedup achieved by GOjDE with different population size.

Population size	F_1			F_2		
	CPU time	GPU time	Speedup	CPU time	GPUtime	Speedup
$N_p = 256$	38.11	13.34	2.86	62.66	13.31	4.71
$N_p = 1024$	160.95	21.05	7.65	259.94	21.13	12.3
$N_p = 2048$	343.02	32.95	10.41	541.27	32.09	16.87
$N_p = 4096$	741.31	60.59	12.23	1141.58	56.81	20.09
	F_3			F_4		
$N_p = 256$	327.7	25.01	13.11	160.59	17.08	9.40
$N_p = 1024$	1323.17	33.63	39.34	652.14	25.36	25.72
$N_p = 2048$	2667.47	45.13	59.11	1325.27	36.86	35.95
$N_p = 4096$	5386.86	71.80	75.03	2704.77	61.38	44.07
	F_5			F_6		
$N_p = 256$	196.94	16.28	12.10	156.56	23.63	6.63
$N_p = 1024$	798.11	24.31	32.83	636.00	31.73	20.04
$N_p = 2048$	1614.47	35.83	45.06	1296.00	44.56	29.08
$N_p = 4096$	3288.05	61.17	53.75	2660.83	70.20	37.90
	F_7			F_8		
$N_p = 256$	66.82	13.27	5.03	36.99	13.55	2.73
$N_p = 1024$	273.57	20.09	13.62	154.52	21.00	7.36
$N_p = 2048$	529.63	24.17	21.91	340.27	32.00	10.63
$N_p = 4096$	1081.26	37.89	28.54	735.30	57.23	12.85
	F_9			F_{10}		
$N_p = 256$	245.92	14.99	16.41	113.78	30.11	3.78
$N_p = 1024$	1015.47	23.38	43.43	471.81	38.08	12.39
$N_p = 2048$	2060.64	35.22	58.51	971.31	54.44	17.84
$N_p = 4096$	4191.53	63.88	65.62	1996.36	83.25	23.98
	F_{11}			F_{12}		
$N_p = 256$	202.81	14.61	13.88	110.34	18.28	6.04
$N_p = 1024$	837.58	22.64	37.00	454.11	25.41	17.87
$N_p = 2048$	1707.86	33.86	50.44	931.05	35.33	26.35
$N_p = 4096$	3482.80	60.76	57.32	1927.80	62.13	31.03
	F_{13}			F_{14}		
$N_p = 256$	285.55	39.81	7.17	202.91	27.03	7.51
$N_p = 1024$	1155.45	47.30	24.43	823.38	35.81	22.99
$N_p = 2048$	2327.25	56.44	41.23	1664.23	45.89	36.27
$N_p = 4096$	4735.14	82.92	57.10	3409.11	72.58	46.97
	F_{15}			F_{16}		
$N_p = 256$	94.11	24.76	3.80	174.52	18.50	9.43
$N_p = 1024$	388.52	40.89	9.50	709.56	25.51	27.81
$N_p = 2048$	799.25	51.38	15.56	1443.91	35.76	40.38
$N_p = 4096$	1662.73	78.22	21.26	2957.83	61.50	48.09
	F_{17}			F_{18}		
$N_p = 256$	298.63	36.03	8.29	271.36	23.13	11.73
$N_p = 1024$	1206.91	43.30	27.87	1097.98	30.83	35.61
$N_p = 2048$	2432.19	53.42	45.53	2213.95	40.72	54.37
$N_p = 4096$	4952.06	80.47	61.54	4518.38	65.98	68.48
	F_{19}					
$N_p = 256$	133.66	33.64	3.97			
$N_p = 1024$	549.30	40.78	13.47			
$N_p = 2048$	1119.81	51.20	21.87			
$N_p = 4096$	2306.03	77.63	29.71			

200, 500 and 1000. We can summarize the experimental results as follows.

- Comparison of GOjDE with DE, CHC, G-CMA-ES and GODE show that GOjDE is better than other four algorithms. The embedded strategies, including self-adapting parameters and GOBL, are helpful to improve the quality of solutions.
- GPU can effectively help GOjDE to reduce the computational time. When the population size is fixed, the obtained speedup tends to decrease with the growth of dimensions.
- Larger population size is helpful to obtain higher speedup. The main reason is that the number of parallelized individuals (threads) increases. That is beneficial for solving higher dimensional problems by setting appropriate population size.

It is applicable to use GPU to solve higher dimensional optimization problems. This will be investigated in our future work.

Acknowledgments

We would like to thank the editor and anonymous reviewers for their detailed and constructive comments that help us to increase the quality of this work. We also thank Mr. Feng Zhu for his implementation. This work was supported by the National Natural Science Foundation of China (Nos 61070008, and 60871021),

Table 13
Mean error values achieved by GOjDE on GPU with different population size, where the best results are shown in *bold*.

Functions	$N_p = 256$ Mean	$N_p = 1024$ Mean	$N_p = 2048$ Mean	$N_p = 4096$ Mean
F_1	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
F_2	4.55E+00	**2.89E+00**	2.91E+00	3.12E+00
F_3	1.40E+02	1.23E+02	1.18E+02	**1.17E+02**
F_4	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
F_5	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
F_6	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
F_7	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
F_8	1.57E+00	**6.94E−01**	1.07E+00	1.23E+01
F_9	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
F_{10}	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
F_{11}	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
F_{12}	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
F_{13}	1.10E+02	1.01E+02	9.86E+01	**9.71E+01**
F_{14}	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
F_{15}	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
F_{16}	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
F_{17}	1.38E+01	6.82E+00	3.37E+00	**2.54E+00**
F_{18}	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**
F_{19}	**0.00E+00**	**0.00E+00**	**0.00E+00**	**0.00E+00**

the Science and Technology Plan Projects of Jiangxi Provincial Education Department (No. GJJ12641), and the Jiangxi Province Soft Science Research Programs (No. 2010ZDR00100).

References

[1] A. Akoglu, G.M. Striemer, Scalable and highly parallel implementation of Smith–Waterman on graphics processing unit using CUDA, Cluster Comput. 12 (2009) 341–352.
[2] A. Auger, N. Hansen, A restart CMA evolution strategy with increasing population size, in: Proceedings of IEEE Congress on Evolutionary Computation, 2005, pp. 1769–1776.
[3] T. Bäck, Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms, Oxford University Publisher, New York, 1996.
[4] W. Banzhaf, S. Harding, W.B. Langdon, G. Wilson, Accelerating genetic programming through graphics processing units, in: Genetic Programming Theory and Practice, vol. VI, 2009, pp. 1–19.
[5] R.E. Bellman, Dynamic Programming, Princeton University Press, 1957.
[6] J. Brest, S. Greiner, G. Bošković, M. Mernik, V. Žumer, Self-adapting control parameters in differential evolution: a comparative study on numerical benchmark problems, IEEE Trans. Evol. Comput. 10 (6) (2006) 646–657.
[7] J. Brest, M.S. Maučec, Self-adaptive differential evolution algorithm using population size reduction and three strategies, Soft Comput. 15 (11) (2011) 2157–2174.
[8] J. Brest, A. Zamuda, B. Bošković, M.S. Maučec, V. Žumer, High-dimensional real-parameter optimization using self-adaptive differential evolution algorithm with population size reduction, in: Proceedings of IEEE Congress on Evolutionary Computation, 2008, pp. 2032–2039.
[9] M. Dorigo, V. Maniezzo, A. Colorni, The ant system: optimization by a colony of cooperating agents, IEEE Trans. Syst. Man Cybern. Part B 26 (1996) 29–41.
[10] A. Duarte, R. Marti, An adaptive memory procedure for continuous optimization, in: Proceedings of International Conference on Intelligent System Design and Applications, 2009, pp. 1085–1089.
[11] L.J. Eshelman, J.D. Schaffer, Real-coded genetic algorithm and interval schemata, Found. Genet. Algorithms 2 (1993) 187–202.
[12] S. García, A. Fernández, J. Luengo, F. Herrera, Advanced nonparametric tests for multiple comparisons in the design of experiments in computational intelligence and data mining: experimental analysis of power, Inform. Sci. 180 (2010) 2044–2064.
[13] S. García, D. Molina, M. Lozano, F. Herrera, A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the CEC'2005 special session on real parameter optimization, J. Heuristics 15 (2009) 617–644.
[14] C. Garcí-Martínez, M. Lozano, Continuous variable neighbourhood search algorithm based on evolutionary metaheuristic components: a scalability test, in: Proceedings of International Conference on Intelligent System Design and Applications, 2009, pp. 1074–1079.
[15] C. García-Martínez, F.J. Rodríguez, M. Lozano, Role differentiation and malleable mating for differential evolution: an analysis on large scale optimisation, Soft Comput. 15 (11) (2011) 2109–2126.
[16] F. Herrera, M. Lozano, D. Molina, Components and parameters of DE, real-coded CHC, and G-CMAES, Technical Report, University of Granada, Spain, 2010.
[17] F. Herrera, M. Lozano, D. Molinam, Test suite for the special issue of Soft Computing on scalability of evolutionary algorithms and other metaheuristics for large scale continuous optimization problems, Technical Report, University of Granada, Spain, 2010. http://sci2s.ugr.es/eamhco/#LSCOP-special-issue-SOCO.
[18] S. Hsieh, T. Sun, C. Liu, S. Tsai, Solving large scale global optimization using improved particle swarm optimizer, in: Proceedings of IEEE Congress on Evolutionary Computation, 2008, pp. 1777–1784.
[19] T. Hu, S. Harding, W. Banzhaf, Variable population size and evolution acceleration: a case study with a parallel evolutionary algorithm, Genet. Program. Evolvable Mach. 11 (2010) 205–225.
[20] J. Kennedy, R.C. Eberhart, Particle swarm optimization, in: Proceedings of IEEE International Conference on Neural Networks, 1995, pp. 1942–1948.
[21] S. Kirkpatrick, C.D. Gelatt, P.M. Vecchi, Optimization by simulated annealing, Science 220 (1983) 671–680.
[22] P. Larranaga, J.A. Lozano, Estimation of Distribution Algorithms—A New Tool for Evolutionary Computation, Kluwer Academic Publishers, Boston, 2001.
[23] J. Liu, J. Lampinen, A fuzzy adaptive differential evolution algorithm, Soft Comput. 9 (6) (2005) 448–462.
[24] D. Molina, M. Lozano, F. Herrera, Memetic algorithm with local search chaining for continuous optimization problems: a scalability test, in: Proceedings of International Conference on Intelligent System Design and Applications, 2009, pp. 1068–1073.
[25] S. Muelas, A. LaTorre, J. Peña, A memetic differential evolution algorithm for continuous optimization, in: Proceedings of International Conference on Intelligent System Design and Applications, 2009, pp. 1080–1084.
[26] nVidia, NVIDIA CUDA programming guide version 2.2.1, 2009.
[27] K. Price, R. Storn, J. Lampinen, Differential Evolution: A Practical Approach to Global Optimization, first ed., Springer-Verlag, New York, 2005.
[28] S. Rahnamayan, H.R. Tizhoosh, M.M.A. Salama, Opposition-based differential evolution algorithms, in: Proceedings of IEEE Congress on Evolutionary Computation, 2006, pp. 2010–2017.
[29] S. Rahnamayan, H.R. Tizhoosh, M.M.A. Salama, Opposition-based differential evolution for optimization of noisy problems, in: Proceedings of IEEE Congress on Evolutionary Computation, 2006, pp. 1865–1872.
[30] S. Rahnamayan, H.R. Tizhoosh, M.M.A. Salama, Opposition-based differential evolution, IEEE Trans. Evol. Comput. 12 (1) (2008) 64–79.
[31] S. Rahnamayan, G.G. Wang, Solving large scale optimization problems by opposition-based differential evolution (ODE), Trans. Comput. 7 (10) (2008) 1792–1804.
[32] D. Robilliard, V. Marion-Poty, C. Fonlupt, Genetic programming on graphics processing units, Genet. Program. Evolvable Mach. 10 (2009) 447–471.
[33] R. Storn, K.V. Price, Differential evolution: a simple and efficient adaptive scheme for global optimization over continuous spaces, ICSI, USA, Tech. Rep. TR-95-012, 1995.
[34] R. Storn, K. Price, Differential evolution—a simple and efficient heuristic for global optimization over continuous spaces, J. Global Optim. 11 (1997) 341–359.
[35] K. Tang, X. Li, S.N. Suganthan, Z. Yang, T. Weise, Benchmark functions for the CEC'2010 special session and competition on large-scale global optimization, Technical Report, Nature Inspired Computation and Applications Laboratory, USTC, China, 2010. http://nical.ustc.edu.cn/cec10ss.php.
[36] K. Tang, X. Yao, P.N. Suganthan, C. Macnish, Y. Chen, C. Chen, Z. Yang, Benchmark functions for the CEC'2008 special session and competition on high-dimensional real-parameter optimization, Technical Report, Nature Inspired Computation and Applications Laboratory, USTC, China, 2007.
[37] H.R. Tizhoosh, Opposition-based learning: a new scheme for machine intelligence, in: Proceedings of International Conference on Computational Intelligence for Modeling Control and Automation, 2005, pp. 695–701.
[38] L. Tseng, C. Chen, Multiple trajectory search for large scale global optimization, in: Proceedings of IEEE Congress on Evolutionary Computation, 2008, pp. 3057–3064.
[39] T. Tušar, B. Filipič, Differential evolution versus genetic algorithms in multiobjective optimization, in: Proceedings of Evolutionary Multi-Criterion Optimization, 2007, pp. 257–271.
[40] L.P. Veronese, R.A. Krohling, Differential evolution algorithm on the GPU with C-CUDA, in: Proceedings of IEEE Congress on Evolutionary Computation, 2010, pp. 1–7.
[41] J. Vesterstrom, R. Thomsen, A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems, in: Proceedings of IEEE Congress on Evolutionary Computation, 2004, pp. 1980–1987.
[42] H. Wang, Y. Liu, S.Y. Zeng, H. Li, C.H. Li, Opposition-based particle swarm algorithm with Cauchy mutation, in: Proceedings of IEEE Congress on Evolutionary Computation, 2007, pp. 4750–4756.
[43] H. Wang, Z.J. Wu, S. Rahnamayan, Enhanced opposition-based differential evolution for solving high-dimensional continuous optimization problems, Soft Comput. 15 (11) (2011) 2127–2140.
[44] H. Wang, Z.J. Wu, S. Rahnamayan, L.S. Kang, A scalability test for accelerated DE using generalized opposition-based learning, in: Proceedings of International Conference on Intelligent System Design and Applications, 2009, pp. 1090–1095.
[45] H. Wang, Z.J. Wu, S. Rahnamayan, Y. Liu, M. Ventresca, Enhancing particle swarm optimization using generalized opposition-based learning, Inform. Sci. 181 (20) (2011) 4699–4714.
[46] M. Weber, F. Neri, V. Tirronen, Shuffle or update parallel differential evolution for large scale optimization, Soft Comput. 15 (11) (2011) 2089–2107.
[47] M.L. Wong, Parallel multi-objective evolutionary algorithms on graphics processing units, in: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference, 2009, pp. 2515–2522.
[48] M.L. Wong, T.T. Wong, Parallel hybrid genetic algorithms on Consumer-Level graphics hardware, in: Proceedings of IEEE Congress on Evolutionary Computation, 2006, pp. 2973–2980.

[49] Z. Yang, K. Tang, X. Yao, Multilevel cooperative coevolution for large scale optimization, in: Proceedings of IEEE Congress on Evolutionary Computation, 2008, pp. 1663–1670.
[50] Z. Yang, K. Tang, X. Yao, Scalability of generalized adaptive differential evolution for large-scale continuous optimization, Soft Comput. 15 (11) (2011) 2141–2155.
[51] J.Q. Zhang, A.C. Sanderson, JADE: adaptive differential evolution with optional external archive, IEEE Trans. Evol. Comput. 13 (5) (2009) 945–958.
[52] S. Zhao, J. Liang, P.N. Suganthan, M.F. Tasgetiren, Dynamic multi-swarm particle swarm optimizer with local search for large scale global optimization, in: Proceedings of IEEE Congress on Evolutionary Computation, 2008, pp. 3846–3853.
[53] S.Z. Zhao, P.N. Suganthan, S. Das, Self-adaptive differential evolution with multi-trajectory search for large scale optimization, Soft Comput. 15 (11) (2011) 2175–2185.
[54] H. Zhou, K.L. Lange, M.A. Suchard, Graphical processing units and high-dimensional optimization. arXiv:1003.3272v1, 2009.
[55] W. Zhu, Massively parallel differential evolution—pattern search optimization with graphics hardware acceleration: an investigation on bound constrained optimization problems, J. Global Optim. 50 (3) (2011) 417–437.

Dr. Hui Wang received his B.Sc. and M.Sc. degrees in computer science from China University of Geosciences in 2005 and 2008. In 2011, he received his Ph.D. degree in computational intelligence from Wuhan University, China. Now, he is a lecture in Nanchang Institute of Technology, China. His research interests include evolutionary computation, large-scale global optimization, and parallel computing. He has published more than 30 international journal/conference papers. He serves as a reviewer for more than ten international journals.

Dr. Shahryar Rahnamayan received his B.Sc. and M.Sc. degrees both with honors in software engineering. In 2007, he received his Ph.D. degree in the field of evolutionary computation from University of Waterloo (UW), Canada. Since August 2007, he has been a chief research manager at OMISA (Omni-Modality Intelligent Segmentation Assistant) Inc. Before joining the faculty of engineering and applied science, University of Ontario Institute of Technology (UOIT), as a faculty member, he was a postdoctoral fellow at Simon Fraser University (SFU) in Canada. His research includes Metaheuristics, evolutionary computation, large-scale optimizations, image processing, and computer vision. He is a reviewer for more than fifteen international journals.

Dr. Zhijian Wu received his B.Sc. degree in mathematics from Jiangxi University in 2005, M.Sc. degree in mathematics from Wuhan University in 1988 and Ph.D.degree in computer science from Wuhan University in 2004. From 2004, he has been a professor at Wuhan University. His research interests include evolutionary computation, large-scale global optimization, parallel computing and inverse problem. He has published more than 70 international journal/conference papers.