Computers & Fluids 110 (2015) 198–210

Contents lists available at ScienceDirect

Computers & Fluids

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

Parallel hybrid PSO with CUDA for lD heat conduction equation

Aijia Ouyang [a], Zhuo Tang [a,*], Xu Zhou [a], Yuming Xu [a], Guo Pan [a], Keqin Li [a,b]

[a] College of Information Science and Engineering, Hunan University, Changsha, Hunan 410082, China
[b] Department of Computer Science, State University of New York at New Paltz, New Paltz, NY 12561, USA

ARTICLE INFO

Article history:
Received 8 December 2013
Received in revised form 6 May 2014
Accepted 15 May 2014
Available online 28 May 2014

Keywords:
Heat conduction equation
Spline difference method
Particle swarm optimization
Conjugate gradient method
GPU

ABSTRACT

Objectives: We propose a parallel hybrid particle swarm optimization (PHPSO) algorithm to reduce the computation cost because solving the one-dimensional (1D) heat conduction equation requires large computational cost which imposes a great challenge to both common hardware and software equipments.

Background: Over the past few years, GPUs have quickly emerged as inexpensive parallel processors due to their high computation power and low price, The CUDA library can be used by Fortran, C, C++, and by other languages and it is easily programmed. Using GPU and CUDA can efficiently reduce the computation time of solving heat conduction equation.

Methods: Firstly, a spline difference method is used to discrete 1D heat conduction equation into the form of linear equation systems, secondly, the system of linear equations is transformed into an unconstrained optimization problem, finally, it is solved by using the PHPSO algorithm. The PHPSO is based on CUDA by hybridizing the PSO and conjugate gradient method (CGM).

Results: A numerical case is given to illustrate the effectiveness and efficiency of our proposed method. Comparison of three parallel algorithms shows that the PHPSO is competitive in terms of speedup and standard deviation. The results also show that using PHPSO to solve the one-dimensional heat conduction equation can outperform two parallel algorithms as well as HPSO itself.

Conclusions: It is concluded that the PHPSO is an efficient and effective approach towards the 1D heat conduction equation, as it is shown to be with strong robustness and high speedup.

1. Introduction

Recently, taking advantage of graphic processing capability, numerical simulation has become available via the utilization of a GPU (graphics processing unit) instead of a CPU (central processing unit) [1]. In its initial stage, however, the GPU cannot be made to communicate with an API (application programming interface) such as OpenGL and DirectX. In November 2006, NVIDIA unveiled the industry's first DirectX 10 GPU, the GeForce 8800 GTX. The GeForce 8800 GTX was also the first GPU to be built with NVIDIAs CUDA (compute unified device architecture). Over the past few years, graphics processing units (GPUs) have quickly emerged as inexpensive parallel processors due to their high computation power and low price [2,3]. The CUDA library can be used by Fortran, C, C++, and by other languages and it is easily programmed. Graphics processing units (GPU) technique has been applied to large-scale social networks [4], smoothed particle hydrodynamics simulations [5], volume visualization [6], hopfield neural network [7], particle filters [8], robotic map [9], finite difference schemes [10,11], object recognition [12], thermal analysis [13], hydrodynamic simulations [14], thermodynamic systems [15], solidification process simulation [16], computational fluid dynamics [17,18], particle simulation [19,20].

The one-dimensional heat conduction equation is a well-known simple second order linear partial differential equation (PDE) [21–24]. PDEs like the heat conduction equation often arise in modeling problems in science and engineering. It is also used in financial mathematics in the modeling of options. For example, the Black–Scholes option pricing model's differential equation can be transformed into the heat conduction equation [25,26].

Over the last few years, many physical phenomena were formulated into nonlocal mathematical model. These physical phenomena are modeled by nonclassical parabolic initial-boundary value problems in one space variable which involve an integral term over the spatial domain of a function of desired solution. This integral term may appear in a boundary condition, which is called nonlocal, or in the governing partial differential equation itself, which is

* Corresponding author.
E-mail address: ztang@hnu.edu.cn (Z. Tang).

http://dx.doi.org/10.1016/j.compfluid.2014.05.020
0045-7930/

often referred to as a partial integro-differential equation, or in both [27].

The presence of an integral term in a boundary condition can greatly complicate the application of standard numerical techniques such as finite difference procedures, finite element methods, spectral techniques, and boundary integral equation schemes. Therefore, it is essential that nonlocal boundary value problems can be converted a more desirable form and to make them more applicable to problems of practical interest.

Recently, nonlocal boundary value problems have been covered in the literature [28–31]. For instance, Dehghan [32] presented a new finite difference technique for solving the one-dimensional heat conduction equation subject to the specification of mass, but their methods are only first-order accurate. Caglar et al. [33] developed a third degree B-spines method to solve the heat conduction Eqs. (1)–(3) with the accuracy of $O(k^2+h^2)$, and at the endpoints, the approximation order is second order only. Liu et al. [34] used a quartic spline method to solve one-dimensional telegraphic equations, we consider the lD heat conduction equation in accordance with the above method. Tsourkas and Rubinsky used a parallel genetic algorithm for solving heat conduction problems [35,36].

It is well-known that genetic algorithm is a classic evolutionary algorithm.It has been successfully applied to all kinds of real problems, such asoptimizing system state in a cloud system [37]. PSO is also an evolutionary algorithm and it is developed by Dr.Eberhart and Dr. Kennedy in 1995, inspired by social behavior ofbird flocking or fish schooling. PSO has an outstandingperformance in optimizing functions. Integrating the successful methods and efficient hybrid strategies, we present a PHPSO algorithm based on CUDA for the one-dimensional heat conduction equation. This paper has some contributions, which are described as follows.

1. A new method for solving the one-dimensional heat conduction equation with the nonlocal boundary value conditions is constructed by using quartic spline functions. It shows that the accuracy of the presented method can reach $O(k^2+h^4)$ at the interior nodal points.
2. We also obtain a new method to deal with the nonlocal boundary conditions with the accuracy being $O(k+h^4)$. It is much better than the classical finite difference method with the accuracy of $O(k+h^2)$.
3. We design a parallel hybrid algorithm by hybridizing the PSO and CGM with the CUDA technique. Such an algorithm can speed up the convergence to the optimum solution.

The remainder of this paper is organized in the following way. Section 2 gives a detailed theoretical analysis and mathematical proof concerning the 1D heat conduction equation. Section 3 shows the process of transforming a system linear systems into an unconstrained optimization problem. In Section 4, the serial algorithms of PSO and CGM are presented. In Section 5, the architecture of CUDA is introduced first, and a parallel PSO algorithm, a parallel CGM algorithm, and a parallel hybrid PSO algorithm are proposed respectively in detail. Section 6 displays the experimental results and discussions. Finally, the paper concludes with Section 7.

2. Method of numerical processing

2.1. Mathematical model

We consider the one-dimensional nonclassical heat conduction equation

$$\frac{\partial u}{\partial t}=\alpha\frac{\partial^2 u}{\partial x^2},\quad 0<x<1,\quad 0<t\leqslant T, \tag{1}$$

with initial conditions

$$u(x,0)=f(x),\quad 0\leqslant x\leqslant 1, \tag{2}$$

$$\frac{\partial u}{\partial x}u(1,t)=g(t),\quad 0<t\leqslant T, \tag{3}$$

and a nonlocal condition

$$\begin{aligned}&\textstyle\int_0^b u(x,t)dx=m(t),\\ &0<t\leqslant T,\quad 0<b<1.\end{aligned} \tag{4}$$

In this problem, $f(x)$, $g(t)$ and $m(t)$ are given functions and α and b are constants. If $b=1$, Eq. (4) can be differentiated as

$$m'(t)=\int_0^1 u_t dx=\int_0^1 \alpha u_{xx}dx=\alpha u_x(1,t)-\alpha u_x(0,t). \tag{5}$$

The derivation holds only when m and u are differentiable.

2.2. Quartic spline and interpolation error

Let Π be a uniform partition of $[0,1]$ as follows

$$0=x_0<x_1<\cdots<x_n=1, \tag{6}$$

where $x_i=ih,\ h=1/n$.

The quartic spline space on $[0,1]$ is defined as

$$S_4^3(\Pi)=\left\{s\in C^3[0,1]:s|_{[x_{i-1},x_i]}\in P_4, i=1(1)n\right\},$$

where P_d is the set of polynomials of degree at most d.

It is easy to know that dim $S_4^3(\Pi)=n+4$. For any $s\in S_4^3(\Pi)$, the restriction of s in $[x_{i-1},x_i]$ can be expressed as

$$s(x)=s_{i-1}+hs'_{i-1}t+h^2s''_{i-1}\frac{t^2}{12}(6-t^2)+h^2s''_i\frac{t^4}{12}+h^3s'''_{i-1}\frac{t^3}{12}(2-t), \tag{7}$$

where $x=x_{i-1}+th,\quad t\in[0,1],\quad s_i=s(x_i),\ s'_i=s'(x_i),\quad s''_i=s''(x_i),\ s'''_i=s'''(x_i),\ i=0(1)n$.

This leads to

$$s_i=s_{i-1}+hs'_{i-1}+\frac{5}{12}h^2s''_{i-1}+\frac{1}{12}h^2s''_i+\frac{1}{12}h^3s'''_{i-1}, \tag{8}$$

$$s'_i=s'_{i-1}+\frac{2}{3}hs''_{i-1}+\frac{1}{3}hs''_i+\frac{1}{6}h^2s'''_{i-1}, \tag{9}$$

$$s'''_i=-s'''_{i-1}+\frac{2}{h}\left[s''_i-s''_{i-1}\right], \tag{10}$$

for $i=1(1)n$. Based on the above three equations, we can have

$$\frac{1}{12}s''_{i+1}+\frac{5}{6}s''_i+\frac{1}{12}s''_{i-1}=\frac{1}{h^2}[s_{i+1}-2s_i+s_{i-1}],\quad i=1(1)\overline{n-1}. \tag{11}$$

Likewise, if we write the quartic spline function s in $[x_{i-1},x_i]$ as follows

$$s(x)=s_i+hs'_it+h^2s''_i\frac{t^2}{12}(6-t^2)+h^2s''_{i-1}\frac{t^4}{12}+h^3s'''_i\frac{t^3}{12}(2+t), \tag{12}$$

where $x=x_i+th,\ t\in[-1,0]$. This leads to

$$s_{i-1}=s_i-hs'_i+\frac{5}{12}h^2s''_i+\frac{1}{12}h^2s''_{i-1}+\frac{1}{12}h^3s'''_i, \tag{13}$$

$$s'_{i-1}=s'_i-\frac{2}{3}hs''_i-\frac{1}{3}hs''_{i-1}+\frac{1}{6}h^2s'''_i, \tag{14}$$

$$s'''_{i-1}=-s'''_i+\frac{2}{h}\left[s''_i-s''_{i-1}\right], \tag{15}$$

for $i=1(1)n$.

Denote $D^i s(x) = s^{(i)}(x)$. We have

$$s_{i\pm1} = s_i \pm h s_i' + \frac{1}{2}h^2 s_i'' \pm \cdots = e^{\pm hD} s_i. \tag{16}$$

Then Eq. (11) can be rewritten as

$$\frac{1}{12}[e^{hD} + 10 + e^{-hD}]s_i'' = \frac{1}{h^2}[e^{hD} - 2 + e^{-hD}]s_i, \quad i = 1(1)\overline{n-1}. \tag{17}$$

Equivalently,

$$s_i'' = \frac{12}{h^2}\frac{e^{hD} - 2 + e^{-hD}}{e^{hD} + 10 + e^{-hD}} s_i, \quad i = 1(1)\overline{n-1}. \tag{18}$$

Theorem 1. *Assume that $g(x)$ is sufficiently smooth in the interval $[0,1]$. Let $s \in S_4^3(\Pi)$ be the quartic spline which interpolates $g(x)$ as follows*

$$s_i = g(x_i), \quad i = 0(1)n, \tag{19}$$

$$\begin{aligned} &s_n'' = g''(x_n), \\ &s_0' + \frac{h^2}{12}s_0''' = g'(x_0) + \frac{h^2}{12}g'''(x_0), \\ &s_n' - \frac{h^2}{12}s_n''' = g'(x_n) - \frac{h^2}{12}g'''(x_n). \end{aligned} \tag{20}$$

Then we have

$$s_i'' = g''(x_i) - \frac{1}{240}h^4 g^{(6)}(x_i) + O(h^6), \quad i = 1(1)\overline{n-1}, \tag{21}$$

$$g_0' = \frac{g_1 - g_0}{h} - \frac{1}{12}h g_1'' - \frac{5}{12}h g_0'' - \frac{1}{12}h^2 g_0''' + O(h^4), \tag{22}$$

$$g_n' = \frac{g_n - g_{n-1}}{h} + \frac{5}{12}h g_n'' + \frac{1}{12}h g_{n-1}'' + \frac{1}{12}h^2 g_n''' + O(h^4), \tag{23}$$

where $g_i = g(x_i),\ g_i' = g'(x_i),\ g_i'' = g''(x_i),\ g_i''' = g'''(x_i),\ i = 0(1)n$.

Proof. The proof of Eq. (21) can been seen in [38].

From Eqs. (11) and (19), we can get

$$(e^{2Dh} + 10e^{Dh} + 1)s_0'' = \frac{12}{h^2}(e^{2Dh} - 2e^{Dh} + 1)g_0''. \tag{24}$$

Then, similar to the proof of Eq. (21), we can obtain the following result

$$s_0'' = g''(x_0) + O(h^4). \tag{25}$$

Then, according to Eqs. (8) and (20), we have

$$\begin{aligned} g_0' + \frac{h^2}{12}g_0''' = s_0' + \frac{h^2}{12}s_0''' &= \frac{s_1 - s_0}{h} - \frac{1}{12}h s_1'' - \frac{5}{12}h s_0'' \\ &= \frac{g_1 - g_0}{h} - \frac{1}{12}h g_1'' - \frac{5}{12}h g_0'' + O(h^4). \end{aligned} \tag{26}$$

Thus, Eq. (22) holds. Similarly, by using Eq. (13), we can prove Eq. (23). □

2.3. Quartic spline method

We consider the following heat conduction Eq. (1) with the initial boundary value conditions (2) and (3) and derivative boundary condition (5).

The domain $[0,1]\times[0,T]$ is divided into an $n \times m$ mesh with the spatial step size $h = 1/n$ in x direction and the time step size $k = T/m$, respectively.

Grid points (x_i, t_j) are defined by

$$x_i = ih, \quad i = 0(1)n, \tag{27}$$

$$t_j = jk, \quad j = 0(1)m, \tag{28}$$

in which n and m are integers.

Let $s(x_i,t_j)$ and U_i^j be approximations to $u(x_i,t_j)$ and $M_i(t) = \frac{\partial^2 s(x,t)}{\partial x^2}\Big|_{(x_i,t)}$, respectively.

Assume that $u(x,t)$ is the exact solution to Eq. (1). For any fixed t, let $s(x,t) \in S_4^3(\Pi)$ be the quartic spline interpolating to $u(x,t)$ as in

$$s(x_i,t) = u(x_i,t), \quad i = 0(1)n, \tag{29}$$

$$\frac{\partial^2 s}{\partial x^2}\Big|(x_n,t) = \frac{\partial^2 u}{\partial x^2}\Big|(x_n,t), \tag{30}$$

$$\frac{\partial s}{\partial x}\Big|(x_0,t) + \frac{1}{12}h^2\frac{\partial^3 s}{\partial x^3}\Big|(x_0,t) = \frac{\partial u}{\partial x}\Big|(x_0,t) + \frac{1}{12}h^2\frac{\partial^3 u}{\partial x^3}\Big|(x_0,t), \tag{31}$$

$$\frac{\partial s}{\partial x}\Big|(x_n,t) - \frac{1}{12}h^2\frac{\partial^3 s}{\partial x^3}\Big|(x_n,t) = \frac{\partial u}{\partial x}\Big|(x_n,t) - \frac{1}{12}h^2\frac{\partial^3 u}{\partial x^3}\Big|(x_n,t). \tag{32}$$

Then it follows from Theorem 1 that

$$\frac{\partial^2 u}{\partial x^2}|(x_i,t) = \frac{\partial^2 s}{\partial x^2}\Big|(x_i,t) + O(h^4), \quad i = 1(1)\overline{n-1}. \tag{33}$$

For any fixed t, by using the Taylor series expansion, Eq. (1) can be discretized at the point (x_i,t_j) into

$$\frac{u(x_i,t_{j+1}) - u(x_i,t_j)}{k} = \frac{1}{2}\alpha\left[\left(\frac{\partial^2 s(x,t)}{\partial x^2}\right)_i^j + \left(\frac{\partial^2 s(x,t)}{\partial x^2}\right)_i^{j+1}\right] + O(k^2 + h^4), \tag{34}$$

where $i = 1(1)\overline{n-1},\ j = 0(1)\overline{m-1}$.

Substituting (34) into spline relation (11) and using Eq. (29), we conclude

$$\begin{aligned} &(1-6\alpha r)u(x_{i+1},t_{j+1}) + (10+12\alpha r)u(x_i,t_{j+1}) \\ &\quad + (1-6\alpha r)u(x_{i-1},t_{j+1}) \\ &= 1 + 6\alpha r)u(x_{i+1},t_j) + (10-12\alpha r)u(x_i,t_j) \\ &\quad + (1+6\alpha r)u(x_{i-1},t_j) + O(k^2+h^4), \end{aligned} \tag{35}$$

where $r = \alpha\frac{k}{h^2}$.

Neglecting the error term, we can get the following difference scheme

$$\begin{aligned} &(1-6\alpha r)U_{i+1}^{j+1} + (10+12\alpha r)U_i^{j+1} + (1-6\alpha r)U_{i-1}^{j+1} \\ &\quad = (1+6\alpha r)U_{i+1}^j + (10-12\alpha r)U_i^j + (1+6\alpha r)U_{i-1}^j, \\ &\quad i = 1(1)\overline{n-1}, \quad j = 0(1)m, \end{aligned} \tag{36}$$

with the accuracy being $O(k^2+h^4)$.

It is evident that the difference scheme (36) identifies the classical fourth order compact difference scheme, which is unconditionally stable.

2.4. The difference scheme on the boundary

From the interpolation conditions (30)–(32) and Theorem 1, for each $t > 0$, we have

$$\begin{aligned} \frac{\partial u(x_0,t)}{\partial x} = \frac{u(x_1,t) - u(x_0,t)}{h} &- \frac{1}{12}h\frac{\partial^2 u(x_1,t)}{\partial x^2} - \frac{5}{12}h\frac{\partial^2 u(x_0,t)}{\partial x^2} \\ &- \frac{1}{12}h^2\frac{\partial^3 u(x_0,t)}{\partial x^3} + O(h^4), \end{aligned} \tag{37}$$

$$\frac{\partial u(x_n,t)}{\partial x}=\frac{u(x_n,t)-u(x_{n-1},t)}{h}+\frac{5}{12}h\frac{\partial^2 u(x_n,t)}{\partial x^2}+\frac{1}{12}h\times\frac{\partial^2 u(x_{n-1},t)}{\partial x^2}+\frac{1}{12}h^2\frac{\partial^3 u(x_n,t)}{\partial x^3}+O(h^4). \tag{38}$$

Let $t=t_{j+1}$, it follows form Eqs. (1) and (37) that

$$u_x(x_0,t_{j+1})=\frac{u(x_1,t_{j+1})-u(x_0,t_{j+1})}{h}-\frac{1}{12\alpha}h\times\frac{u(x_1,t_{j+1})-u(x_1,t_j)}{k}-\frac{5}{12\alpha}h\times\frac{u(x_0,t_{j+1})-u(x_0,t_j)}{k}-\frac{1}{12}h^2\frac{\partial^3 u(x_0,t_{j+1})}{\partial x^3}+O(k+h^4). \tag{39}$$

From Eqs. (1), (3) and (5), we can get

$$\frac{\partial^3 u(0,t)}{\partial x^3}=\frac{1}{\alpha}\frac{\partial^2 u(0,t)}{\partial x\partial t}=\frac{1}{\alpha^2}m''(t)-\frac{1}{\alpha}g'(t). \tag{40}$$

Substituting (40) into (39) and neglecting the error term, we get the fourth-order difference scheme at $x=x_0$

$$\left(1-\frac{1}{12r}\right)U_1^{j+1}-\left(1+\frac{5}{12r}\right)U_0^{j+1}+\frac{1}{12r}U_1^j+\frac{5}{12r}U_0^j=hu_x(0,t_{j+1})+\frac{1}{12}h^3\left(\frac{1}{\alpha^2}m''(t_{j+1})-\frac{1}{\alpha}g'(t_{j+1})\right). \tag{41}$$

Similarly, the difference scheme at the other end $x=x_n$ is

$$\left(1+\frac{5}{12r}\right)U_n^{j+1}+\left(-1+\frac{1}{12r}\right)U_{n-1}^{j+1}-\frac{5}{12r}U_n^j+\frac{1}{12r}U_{n-1}^j=hu_x(1,t_{j+1})-\frac{1}{12\alpha}h^3g'(t_{j+1}). \tag{42}$$

3. Derivation of objective function

The formula of a system linear systems is shown as follows:

$$AX=B, \tag{43}$$

or

$$\sum_{j=1}^{N}a_{ij}x_j=b_i;\quad i=1,2,\ldots,M. \tag{44}$$

In the above formula, $A=(a_{ij})_{M\times N}\in R^{M\times N}$ is a matrix, $X=(x_1,x_2,\ldots,x_N)^T$ is a vector, $B=(b_1,b_2,\ldots,b_M)^T$ is also a vector.

Set the absolute error of X and B be respectively δX and δB. So the relative error of X can be estimated as follow:

$$\|\delta X\|/\|X\|\leqslant K(A)\|\delta B\|/\|B\|. \tag{45}$$

If we obtain the L_2 norm for the above formula, the condition number of matrix A is:

$$K(A)=\sigma_{max}/\sigma_{min}. \tag{46}$$

In the formula, σ_{max} and σ_{min} is respectively the maximum and minimum of singular value of matrix A. If $K(A)$ is very large, the formula (43) is morbid state linear equation group. The characteristic of morbid state linear equation group is: even though A and B are little changed, the solution of them will be greatly different.

To use PSO to solve linear equation group (43), we transform the linear equation group problem into an unconstrained optimization problem. Obviously, the L_1 norm of linear equation group (43) is the optimal solution of the problem (47). Both of them are completely equivalent. The formula of the problem is shown as follows:

$$min\ E=min\sum_{i=1}^{m}\sum_{j=1}^{N}|a_{ij}x_j-b_i|. \tag{47}$$

Consequently, we set the fitness function as follows:

$$f(x)=min\sum_{i=1}^{m}\sum_{j=1}^{N}|a_{ij}x_j-b_i|, \tag{48}$$

in formula (48), The solution vector $X=(x_i,x_2,\ldots,x_n)$ is composed by $x_j(j=1,2,\ldots,N)$. X is a real vector.

4. Serial algorithms

4.1. Particle swarm optimization

Particle swarm optimization (PSO), a swarm intelligence algorithm to solve optimization problems, is initialized randomly with a cluster of particles, after which it searches for optimal value by means of updating generations [39]. Given that the search space is D-dimensional, the population is composed of many particles. Particle i refers to the i-th particle in the population, which is denoted by D-dimensional vector $X_i=(x_{i1},x_{i2},\ldots,x_{iD})^T$. V denotes the search velocity of particles, and $V_i=(v_{i1},v_{i2},\ldots,v_{iD})^T$ reveals the search velocity of particle i. The visited positions of particles are denoted by P, and $P_i=(p_{i1},p_{i2},\ldots,p_{iD})^T$ reveals local best position, and it is the best previously visited position of particle i. The global best position is represented by $G=(g_1,g_2,\ldots,g_D)^T$, which shows the optimal of all local best positions among the population. At each step, the search velocity of all particles and their local best position will be given a value according to Eqs. (49) and (50), the process goes repeatedly till a user-defined stopping criterion is met.

$$v_{id}^{k+1}=\omega v_{id}^k+c_1r_1\left(p_{id}^t-x_{id}^k\right)+c_2r_2\left(g_{id}^k-x_{id}^k\right), \tag{49}$$

$$x_{id}^{k+1}=x_{id}^k+v_{id}^{k+1}. \tag{50}$$

Here, k indicates the current iterative step, $i\in\{1,2,\ldots,N\}$, and $d\in\{1,2,\ldots,D\}$, N refers to the number of the total populations. r_1 and r_2 represent independently uniformly distributed random variables with a range of $(0,1)$, c_1 and c_2 denote the acceleration constants, and ω refers to the inertia weight factor [40].

4.2. Conjugate gradient method

The CGM is one of the most popular iterative methods dedicated to solving linear systems of equations with sparse, symmetric and positive-definite matrix of coefficients. A version of the CGM algorithm implemented in this study is shown in Algorithm 1.

The main steps of CGM for an unconstrained optimization problem are as follows:

Step 1. Set a initial point $x^{(0)}$, and precision definition $\varepsilon>0$.
Step 2. If $\|\nabla f(x^{(0)})\|\leqslant\varepsilon$, stop computation, the minimum point is $x^{(0)}$, otherwise, go to Step 3.
Step 3. Let $p^{(0)}=-\nabla f(x^{(0)})$, and set $k=0$.
Step 4. Use one dimensional search to solve t_k, obtain $f(x^{(k)}+t_kp^{(k)})=\min f(x^{(k)}+tp^{(k)})$, $t\geqslant 0$, let $x^{(k+1)}=x^{(k)}+t_kp^{(k)}$, go to Step 5.

Step 5. If $\|\nabla f(x^{(k+1)})\| \leqslant \varepsilon$, stop computation, the minimum point is $x^{(k+1)}$, otherwise, go to Step 6.
Step 6. If $k+1=n$, let $x^{(0)}=x^{(n)}$, go to Step 3, otherwise, go to Step 7.
Step 7. Let $p^{(k+1)}=-\nabla f(x^{(k+1)})+\lambda_k p^{(k)}$, $\lambda_k=\frac{\|\nabla f(x^{(k+1)})\|}{\|\nabla f(x^{(k)})\|}$, let $k=k+1$, go to Step 4.

Algorithm 1. Conjugate gradient method

Input: A, b, n
Output: x_{k+1}
$r_0 := b - Ax_0$
$p_0 := r_0$
$k := 1$
while $k <= n$ **do**
  $\alpha_k := \frac{r_k^T r_k}{P_k^T A P_k}$
  $x_{K+1} : x_k + \alpha_k P_k$
  $r_{K+1} : r_k - \alpha_k A P_k$
  if $r_{k+1} < m$ **then**
    $\beta_k := \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
    $p_{k+1} := r_{k+1} + \beta_k P_k$
    $k := k+1$
  end if
end while
Output solution

4.3. Hybrid particle swarm optimization

The global search ability of the basic PSO is extremely strong while the local search ability is extremely weak, which is exactly the opposite of the CGM. To overcome the shortcomings of the two algorithms and utilize only their advantages, we present a hybrid particle swarm optimization (HPSO) algorithm, in which the CGM is incorporated into the basic PSO. In the HPSO, because the CGM is a local algorithm, it can obtain the optimal value with no need to calculate the derivative of fitness function; therefore, based on the original PSO, only a decent computation cost is added. In addition, each particle in the population is regarded as a point of the CGM and each iteration of HPSO has two parts: the CGM part and PSO part. In each iteration, all particles are updated, and the elite individual generated by PSO algorithm will be replaced by the best individual generated by CGM. In this way, the accuracy of searching the position in each HPSO iteration is improved. Therefore, the HPSO not only has higher precision but also helps to obtain a faster convergent speed compared with the original PSO and CGM.

The main steps of HPSO for an unconstrained optimization problem are as follows:

Step 1. Initialize the parameters of population, such as, population size, position, velocity, acceleration factor, and inertia weight factor.
Step 2. Compute the fitness value of each particle.
Step 3. Find the local optimum and the global optimum.
Step 4. Update the velocity and position of each particle.
Step 5. Compute the fitness value of population.
Step 6. Update the local optimum and the global optimum.
Step 7. Execute a local search precisely by CGM for the individual of global optimum in the population of HPSO.
Step 8. If the termination criteria are satisfied, output results and terminate the HPSO algorithm; otherwise, go to **Step 4**.

Algorithm 2. Particle swarm optimization

Input: $x_{min}, x_{max}, G, D, N, p, lbp, gbp, f, lbf, gbf, v$
Output: gbp', gbf'
Initialize parameters p, lbp, and v
Compute fitness f
Find lbp and gbp
for $g = 1 : G$ **do**
  Update v and p
  Evaluate fitness f
  Update lbp and gbp
end for
Output gbp and gbf

Algorithm 3. Hybrid particle swarm optimization

Input: $x_{min}, x_{max}, G, D, N, p, lbp, gbp, f, lbf, gbf, v$
Output: gbp', gbf'
Initialize parameters p, lbp, and v
Compute fitness f
Find lbp and gbp
for $g = 1 : G$ **do**
  Update v and p
  Evaluate fitness f
  Update lbp and gbp
  Execute local search of CGM
end for
Output gbp and gbf

5. Parallel algorithms

5.1. The architecture of CUDA

We describe in brief the CUDA architecture in this subsection; by this means, we hope that to comprehend the design of our presented GPU implementation can become easier. A GPU is a multi-core, multi-threaded highly parallel processor with enormous computing power. The NVIDIA Corporation released the CUDA framework to help users develop software for their GPUs. The architecture of CUDA is designed around a scalable array of multi-processors as shown in Fig. 1, which is based on a SIMT programming

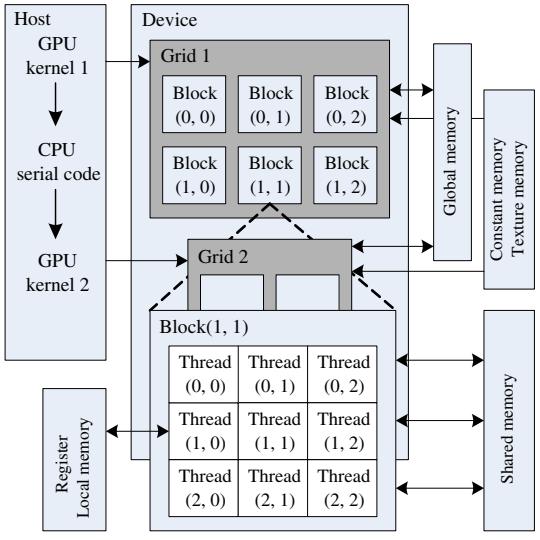

Fig. 1. The architecture of CUDA.

Table 1
Specifications of GPU.

Parameter	GTX 465	C2050
Number of stream processors	352	448
Global memory size (GB)	1	2.62
Shared memory size (KB)	48	48
Constant memory size (KB)	64	64
Memory bus width (bits)	256	384
Memory bandwidth (GB/s)	102.6	144
Memory clock rate (MHz)	802	1536
Estimated peak performance (Gflops)	855.4	1030
GPU clock rate (MHz)	0.6	1.15
Warp Size	32	32
Maximum number of threads per MP	1024	1024
CUDA compute capability	2.0	2.0

modal. A parallel program on a GPU (also termed device) is interleaved with a serial one executed on the CPU (also termed host). A thread block has a stream of threads executed concurrently on a multiprocessor, which has synchronized access to a shared memory. However, threads originating from different blocks cannot be cooperated in this way. These threads are combined to form a grid. Threads in a grid are enumerated and distributed to multiprocessor when a CUDA code on the CPU calls a kernel function. All the threads in the grid can access the same global memory. Two additional read-only memory spaces are accessible by all the threads, the constant memory and texture one (see Table 1).

5.2. Parallel particle swarm optimization

In this section, we would like to explain the meaning of each kernel function in detail with its pseudo code attached. There are five kernel functions in the PPSO algorithm, and they are implemented in GPU. The flow chart of PPSO is shown in Fig. 2.

5.2.1. Kernel function dev_rnd

The purpose of *dev_rnd* is to generate a random number sequence of initial state. Each thread is a complex variable, which needs to contain curand library files and *curand_kernel*, and call the library function *curand_init* to generate high quality pseudo-random number. Given the differenced between seeds, *curand_init* would produce different initial states and sequence. Each thread is assigned a unique serial number.

Sequences generated with different seeds usually do not have statistically correlated values, but some choices of seeds may result in statistically correlated sequences. Sequences generated with the same seed and different sequence numbers will not have statistically correlated values. For the highest quality parallel pseudo-random number generation, each experiment should be assigned a unique seed. Within an experiment, each thread of computation should be assigned a unique sequence number. If an experiment spans multiple kernel launches, it is recommended that threads between kernel launches be given the same seed, and sequence numbers be assigned in a monotonically increasing way. If the same configuration of threads is launched, random state can be preserved in global memory between launches to avoid state setup time [41].

Experiments have shown that the random function in *curand_library* has better quality and higher efficiency than that generated by the rand function of CPU.

5.2.2. Kernel function init

Init is used to initialize a particle position, velocity, and the individual best position in populations according to the formula in the PSO. Every thread in thread blocks represents a complex variable. Each row in the grid represents a particle. After defining x_{id} dimension index, index of population y_{id}, thread (namely, each position variable) position in the whole grid *posID*, we call *curand_uniform*() to generate two random numbers stochastically and uniformly distributed between 0.0 and 1.0, and then use them to initialize the position and velocity respectively.

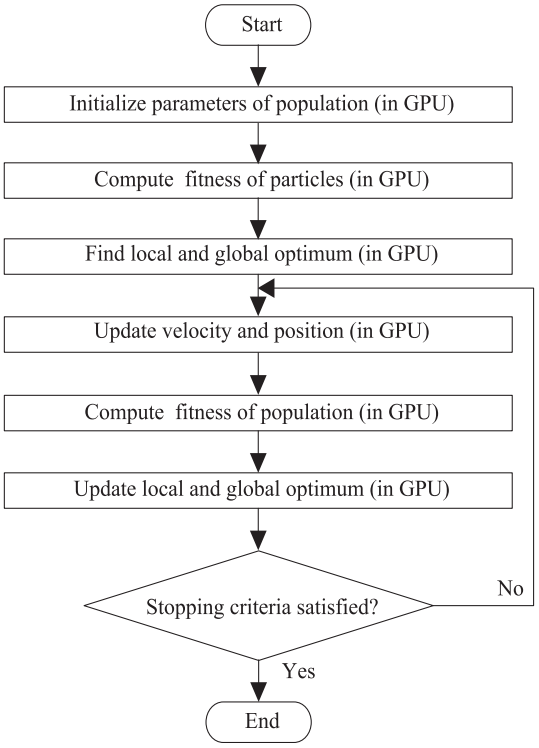

Fig. 2. Flowchart of PPSO.

5.2.3. Kernel function Update

Update is used to modify the position, individual optimal position and velocity of each particle in the population for each thread. As for the parameter setting, it is the same as *init*. It firstly updates the individual optimal position according to the modifying mark, and then respectively calls *curand_uniform* to generate two random numbers between two 0.0 and 1.0 respectively. They are used to generate a new particle velocity, further update particle position, and reinitialize the numbers exceeding the limits at the same time.

5.2.4. Kernel function Findgbf

Findgbf is used to update the individual optimal position, and find the global optimal index and global optimal fitness. A particle is a thread. In this paper, the population numbers are set less than 512, thus a thread block is a whole population. We give full play to the advantages of shared memorizer and registers. First, we need to define the thread *pid* and shared memorizer variable *s* and register variable *t*, and then initialize *s* and *t* in accordance with different requirements. If the initial mark $flag = 0$, the fitness value *f* should be copied directly to individual extremum *lbf*, otherwise, *lbf* and modifying mark *g_u* should be modified correspondingly after comparison. Then we need to call device function $reduceToMax(s, pid)$ or $reduceToMin(s, pid)$ to operate and summarize according to the maximum or minimum value for parallel reduction, in order to find the group extremum *gbf*, and ultimately the global optimal index *ID*. The skill is that we should set *t* to the maximum number of threads that are not the group extremum *gbf*, otherwise *t* would be the thread number of *gbf*. Then we should reuse shared memorizer and call $reduceToMin(s, pid)$ to execute the reduction then assign the first thread to *ID*. If the most optimal value of this generation is not superior to group optimal value *gbf*, *ID* would be *P*, which means there is no need to update the best position *gp* of

the group. Parallel reduction *reduceToMax* or *reduceToMin*. The first thing is to perform the first-level reduction (parallel reading data), then record them in a shared memorizer, and finally perform the second-level reduction in the shared memorizer. Since the execution efficiency of the loop on the CPU is not high, the optimization seems very important. The full loop unrolling, the code optimization by a template parameter, and the unnecessary synchronization in a warp greatly improve the reduction efficiency.

5.2.5. Kernel function Findgbp

It is required by *Findgbp* to find best position *gbp* for groups according to the global optimal index. A thread is a complex variable, and the size of dimension *y* of the grid is 1, which means that if global optimal index *ID* is not equivalent to *p*, the *ID* line of the position of the only particle would be copied to the best position *gbp* for groups. These functions attached by detailed pseudo code. Experiments have proved that it is very effective to combine the particle swarm with conjugate gradient algorithm to implement the GPU parallel, which greatly improves the search quality and computing speed, as well as the performance of algorithm. In kernel function executive parameter, we define *block_xsize* (not more than 512; 128 or 256 is more appropriate; at the same time, SM usually have enough resources to perform at least two active blocks), $block\_ysize = 1, grid\_ysize = p$, namely the number of populations, $grid\_xsize = d + block\_xsize - 1)/block\_xsize$, to represent the block dimensions by this form, to ensure that the number of blocks is an integer, and the number of threads in the whole grid is more than the actual amount of elements to process. Accordingly, in kernel, *if* $(xid < d)$ which can skip the redundant data is used to calculate. If the minimum value is needed, we define better as "<", and otherwise ">".

Algorithm 4. Parallel PSO algorithm in GPU

Input: $G, D, g\_D, flag, N, p, lbp, gbp, h\_gbp, f, lbf, gbf, h\_gbf, v$, *dsg_u*, *ID*, *Grid*, *Block*, *gbfGrid*, *gbfBlock*, *gbpGrid*, *gbpBlock*
Output: *gbp*, *gbf*
1: Initialize the parameters of device
2: Initialize the numbers of random in CUDA software: *dev_rnd*⋘*Grid*, *Block*⋙(*ds*, *time (NULL)*)
3: Initialize population parameters and compute fitness *f*: *init*⋘*Grid*, *Block*⋙(*p*, *lbp*, *v*, *ds*), *h_cf*(*p*, *f*, *N*, *D*)
4: Update optimum *lbp* and find optimum *gbp*: *Findgbf*⋘*gbfGrid*, *gbfBlock*⋙(*f*, *lbf*, *gbf*, *g_u*, *ID*, *flag*), *Findgbp*⋘*gbpGrid*, *gbpBlock*⋙(*gbp*, *p*, *ID*)
5: **for** $g = 1$ to G **do**
6: $flag = 1$
7: Update the velocity and position of each individual in accordance with Eqs. (49) and (50), *Update*⋘*Grid*, *Block*⋙(*p*, *lbp*, *gbp*, *v*, *g_u*, *ds*)
8: Compute the fitness *f*: *h_cf*(*p*, *f*, *N*, *D*)
9: Update individual best position *lbp*, and find global best individual *gbp*
10: **end for**
11: Return the global optimal position *gbp* and fitness *gbf*: *cudaMemcpy*(*h_gbp*, *gbp*, *D***sizeof*(float), *cudaMemcpyDevieToHost*), *cudaMemcpy*(*h_gbf*, *gbf*, *sizeof*(float), *cudaMemcpyDevieToHost*)
12: *printResults*()
13: *writedata*()
14: *H_Free*()
15: *cudaThreadExit*()

Algorithm 5. Random numbers of CUDA

Kernel function: *dev_rnd*⋘*Grid*, *Block*⋙(*ds*, *seed*))
Define executive parameters
dim3 *Block*(*block_xsize*, *block_ysize*,1)
dim3 *Grid*(*grid_xsize*, *grid_ysize*)
Input: *seed*
Output: *ds*
1: Define the parameters: dimension index x_{id}, population index y_{id}, thread position *posID*
2: **if** $x_{id} < d$ **then**
3: Invoke *curand_init*(*seed*, *posID*, 0, &*ds*[*posID*]) to generate random numbers in CUDA
4: **end if**

Algorithm 6. Initialize population with CUDA

kernel function: *init*⋘*Grid*, *Block*⋙(*p*, *lp*, *v*, *ds*)
Input: *ds*
Output: p, lbp, v,
1: Define the dimension index x_{id}, group index y_{id}, the thread position in the grid *posID*
2: **if** $x_{id} < d$ **then**
3: Load the sequence of the initialization status of the random number *ds* from global memory and also assign it to the register variable *ls*
4: Invoke *curand_uniform*(&*ls*) to generate the random number r_1 and r_2 between 0.0 and 1.0
5: Store the *ls* back into the global memory variable *ds*
6: Generate register variable *x* and *vel* based on the velocity and position equation
7: Copy *x* back into *p* and *lbp*; copy *vel* back into *v*
8: **end if**

Algorithm 7. Update of velocity and position

Kernel function: *Update*⋘*Grid*, *Block*⋙(*p*, *lbp*, *gbp*, *v*, *g_u*, *ds*)
Input: p, lbp, gbp, v, g_u, ds
Output: p', lbp', v'
1: Define the dimension index x_{id}, group index y_{id}, the thread position in the grid *posID*
2: **if** $x_{id} < d$ **then**
3: Load p, lbp, gbp, v, g_u, and *ds* from the global memory and also assign them to register variables *pos*, lb, gb, vel, u and *ls*
4: Invoke *curand_uniform*(&*ls*) to generate the random number *r*1 and *r*2 between 0.0 and 1.0
5: Store the *ls* back into the global memory variable *ds*
6: **if** $u == 1$ **then**
7: Assign *pos* to *lb* and *lbp*
8: **end if**
9: Update the velocity based on Eq. (49) and copy it back to *v*
10: Update the particle position based on Eq. (50) and copy it back to *p*
11: **end if**

Algorithm 8. Update of local and global optimum

Kernel function: *Findgbf*⋘*gbfGrid*, *gbfBlock*⋙(*f*, *lbf*, *gbf*, *g_u*, *ID*, *flag*)
Define executive parameters
dim3 *gbfGrid*(1, 1)
dim3 *gbfBlock*(dev_P, 1, 1)
Input: $f, lbf, gbp, v, g\_u, ds, flag$
Output: f', lbf', gbf, g_u', ID
1: Define the thread or particle index t_{id}
2: Initialize the shared memory vector *s* and register variable *bf*. To get the maximum value, it will be valued as 0. To get the minimum value, it will be valued as 1E−20
3: **if** $t_{id} < N$ **then**
4: Load *f* from global memory and assign it to *s* and *bf*
5: Synchronization
6: **if** $flag == 0$ **then**
7: Copy the *bf* to the individual best fitness *lbf*
8: **else**
9: Load *lbf* from global memory and assign it to the register variable *bf*1
10: Assign the "*bf* better *bf*1" to register variable update and copy it back to *g_u*
11: **if** $update == 1$ **then**
12: Replace the individual optimum using new fitness
13: **end if**
14: **end if**
15: **if** *MAXIMIZE* **then**
16: $reduceToMax < dev\_N > (s, t_{id})$
17: **else**
18: $reduceToMin < dev\_N > (s, t\_id)$
19: **end if**
20: **if** $flag == 0 || (flag == 1 \&\& s[0] better * gbf)$ **then**
21: **if** $t\_id == 0$ **then**
22: Assign the best fitness to the global best value
23: **end if**
24: Assign thread index of the best value to the *bf*, or the *bf* valued as dev_N; Assign the *bf* to the shared memory variable *s*
25: $reduceToMin < dev\_N > (s, t\_id)$
26: Copy the best value index *s*[0] back into the *ID*
27: **else**
28: $ID = N$
29: **end if**
30: **end if**

Algorithm 9. Search global optimal position

Kernel function: *Findgbp*⋘*gbpGrid*, *gbpBlock*⋙(*gbp*, *p*, *ID*)
Define executive parameters
dim3 *gbpBlock*(*block_xsize*, 1, 1)
dim3 *gbpGrid*(*grid_xsize*, 1)
Input: p, ID, d
Output: *gbp*
1: Define the dimension index x_{id}
2: **if** $x_{id} < D$ && $ID! = N$ **then**
3: Assign the solution of the No. *ID* particle in the *p* to the global best position *gbp*
4: **end if**

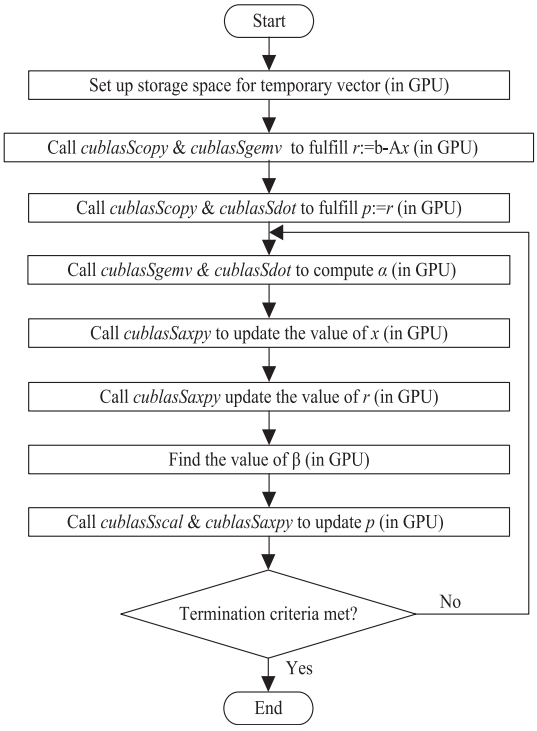

Fig. 3. Flowchart of PCGM.

5.3. Parallel conjugate gradient method

Conjugate gradient is a method which lies between the steepest descent method and Newton method. It only needs first derivative information and overcomes the slow convergence of steepest descent method. Moreover, it does not need to store and calculate Hesse matrix and inverse of Newton method. The characters including the fast convergence speed and quadratic termination make it one of the most efficient algorithms to solve large linear equations. It needs small storage capacity, has step convergence, high stability, and does not require any external parameters. To realize GPU of conjugate gradient requires CUBLAS library, that is, the header file needs to contain cublas.h. Assume that we use it to obtain the solution of linear equations $Ax = b$, the pseudo code is shown as Algorithm 10.

All operations are completed in the GPU. In the first place, the distribution of the temporary variables is conducted in the video memory of GPU, so *cublasAlloc*() and *cublasFree*() are used to replace *malloc*() and *free*() in standard C library. And all the vector and matrix operations adopt the functions in CUBLAS library. The role of each function is explained as follows. *CublasAlloc* is to allocate the video memory space; *CublasFree* is to release the video memory space; This function of *cublasScopy* copies the vector *x* into the vector *y*; whist the function of *cublasSdot*: computes the dot product of vectors *x* and *y*; *cublasSgemv* is to performs the matrix–vector multiplication; the function of *cublasSaxpy* multiplies the vector *x* by the scalar and adds it to the vector *y* overwriting the latest vector with the result; with regards to the function of *cublasSscal*, it mainly scales the vector *x* by the scalar and overwrites it with the result (see Fig. 3).

Algorithm 10. Parallel conjugate gradient method

Input: n, m, A, b, x
Output: X
1: Define variables $\alpha, \beta, \varepsilon, r2$ temporary vectors p, r, AP. Invoke the instruction *cublasAlloc* to distribute storage
2: Invoke *cublasScopy* to achieve the first step: $r := b - Ax$
3: Invoke *cublasScopy*(n, r, 1, p, 10), *cublasSdot*(n, r, 1, emphr, 1) to achieve the second step: $p := r$
4: **for** $k = 1$ to n do **do**
5: Assign the value of ε to r2
6: Invoke *cublasSgemv*('n', n, 1.0f, A, n, p, 1, 0.0f, Ap, 1);
7: Solve the value of $\alpha, \alpha = r2/cublasSdot$;
8: Invoke *cublasSaxpy*(n, α, p, 1, x, 1) to achieve the sixth step and update x
9: Invoke *cublasSaxpy*(n, $-\alpha$, Ap, 1, r, 1) to achieve the seventh step and update r
10: **if** $\varepsilon = cublasSdot < m$ **then**
11: break;
12: **end if**
13: Achieve the ninth step, obtain $\beta, \beta = \varepsilon/r2$
14: Invoke *cublasSscal*(n, β, p, 1) and *cublasSaxpy*(n, 1.0f, r, 1, p, 1) to achieve the tenth step and update p.
15: **end for**
16: Invoke the instruction *cublasFree* to free the memory of p, r and Ap
17: Realize the thirteenth step and obtain x.

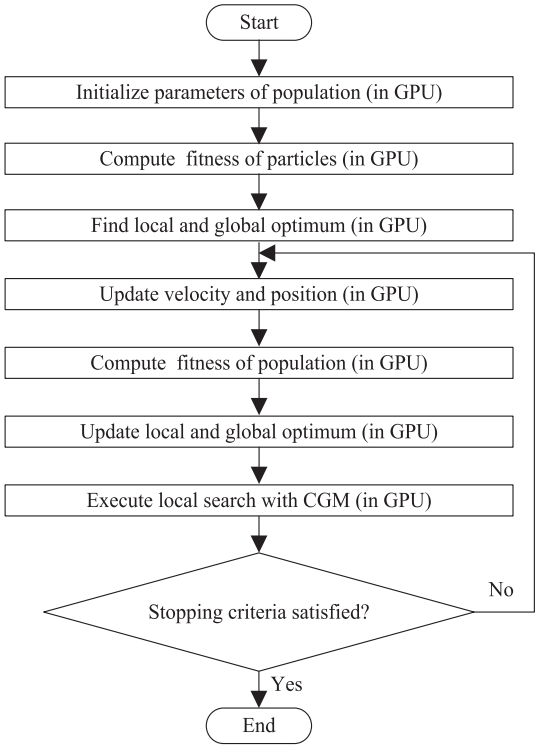

Fig. 4. Flowchart of PHPSO.

5.4. Parallel hybrid particle swarm optimization

Both PSO and CGM have strong parallelism. Operating the combination of them on GPU can improve the search quality and efficiency. GPU program needs to take into consideration algorithm, parallel partition, instruction stream throughput, memory bandwidth and many other factors. First, we need to confirm the serial and parallel part of the task. In the PHPSO algorithm, all the steps in the process can be implemented on GPU. Then we need to initialize the device and prepare data. We store all vectors in the global memorizer and constants in a constant cache memorizer to optimize the memory and meet the need of merger access. We have to make sure that the first visiting address starts from 16 integer times, and the word length of data read each time in every thread is 32 bit. Finally, each step is mapped to a kernel function suiting CUDA two parallel layers model and an instruction stream would be optimized. If only a small amount of thread is needed, an approach similar to "if $thread < N$" could be used to avoid the long time cost or incorrect results caused by the multiple threads running simultaneously. To speed up, quick instructions that adopt CUDA arithmetic instruction to centralize could be used. Using $\#unroll$ could let the compiler develop loops effectively. We should try to avoid bank conflict, balance the resources, and adjust the usage of shared memory and the register. We can regard particles, that is, individuals as threads, or the complex variables as threads. Therefore, we can use a complex variable as a thread to initialize and update the population to improve parallelism. CUDA runtime API or CUDA driver API can be used to manage GPU resources, allocate the memory and start the kernel function on GPU. In this paper, we use the runtime API to implement operations, such as equipment management, context management, and executive control. The detailed processing flow is shown as Fig. 4.

Algorithm 11. Parallel hybrid PSO algorithm in GPU

Input:
$x\_min, x\_max, G, D, g\_D, N, flag, ds, p, lbp, v, f, lbf, g\_u, gbp, gbf, ID, A, b, x, n, m, Grid, Block, gbfGrid, gbfBlock, gbpGrid, gbpBlock$
Output: gbp, gbf
1: Initialize the parameters of device
2: Initialize the number of random in CUDA: *dev_rnd*$\lll Grid, Block \ggg$(ds, $seed$)
3: Initialize population parameters and compute fitness f: *init*$\lll Grid, Block \ggg$(p, lbp, v, ds), *h_cf*(p, f, N, D)
4: Update optimum lbp and find optimum gbp: *Findgbf*$\lll gbfGrid, gbfBlock \ggg$($f$, lbf, gbf, g_u, ID, $flag$), *Findgbp*$\lll gbpGrid, gbpBlock \ggg$($gbp$, p, ID)
5: **for** $g = 1$ to G **do**
6: $flag = 1$
7: Update the velocity and position of each individual in accordance with Eqs. (49) and (50), Update$\lll Grid, Block \ggg$(p, lbp, gbp, v, g_u, ds)
8: Compute the fitness f: *h_cf*(p, f, N, D)
9: *Findgbf*$\lll gbfGrid, gbfBlock \ggg$($f$, lbf, gbf, g_u, ID, $flag$);
10: **end for**
11: Return the global optimal position gbp and fitness gbf: *cudaMemcpy*(*h_gbp*, gbp, $D^*sizeof(float)$, *cudaMemcpyDevieToHost*), *cudaMemcpy*(*h_gbf*, gbf, $sizeof(float)$, *cudaMemcpyDevieToHost*)
12: *printResults*()
13: *writedata*()
14: *H_Free*()
15: *cudaThreadExit*()

6. Experimental results and discuss

6.1. Example of application

In this section, our new method is tested on the following problems selected the literature [32]. Absolute errors of numerical solutions are calculated and compared with those obtained by using the three degree B-spline method [33].

In this example, we consider the following heat conduction equation

$$f(x) = \cos\left(\frac{\pi}{2}x\right), \quad 0 < x < 1, \tag{51}$$

$$g(t) = -\exp\left(-\frac{\pi^2}{4}t\right), \quad 0 < x < 1, \tag{52}$$

$$m(t) = \frac{4}{\pi^2}\exp\left(-\frac{\pi^2}{4}t\right), \quad 0 < x < 1, \tag{53}$$

with

$$u(x,t) = \exp\left(-\frac{\pi^2}{4}t\right)\cos\left(\frac{\pi}{2}x\right), \tag{54}$$

as its analytical solution.

6.2. Parameter setting

All the experiments are carried out on an eight-core 2.3 GHz AMD Opteron 6134 machine with 8 GB main memory, using the GCC compiler under Linux. The parameter setting of PHPSO in terms of population size and stopping criteria is the same as PSO, HPSO and PPSO to ensure fairness of comparison. The population size of the five evolutionary algorithms is set as to 16, and each of the evolutionary algorithms is stopped after 1000 generations. The other parameters of PSO, HPSO, PPSO and PHPSO are set as follows: the maximum velocity $V_{max} = 1.5$, acceleration factor $C_1 = C_2 = 1.2$, inertia weight factor $\omega = 0.9$. Therefore, in these experiments, all the methods are run on the same computer configuration for one-dimensional heat conduction equation. Twenty independent runs with different methods are performed for different years and the average values are recorded in the corresponding tables, with the best solution marked in bold.

6.3. Performance of HPSO

At first, the results with $h = k = 0.05$, $h = k = 0.025$, $h = k = 0.01$, $h = k = 0.005$, $h = k = 0.0025$, $h = k = 0.001$, using HPSO discussed in Sections 3 and 4, are shown in Table 2. We present the relative error $\text{abs}(U_i^j - u(x_i, t_j))/\text{abs}(u(x_i, t_j))$ for $u(0.5, 0.1)$ using HPSO. The numerical results are compared with those results obtained by the methods in [42–45], see Table 2. It can be seen from Table 2 that the numerical results of the proposed HPSO are much better than that in [42–45]. In [32], Dehghan solved the example by using the Saulyev I's technique with the accuracy being first-order with respect to the space and time variables, and the numerical integration trapezoidal rule is used to approximate the integral condition in (4). However, in this paper, we present a new method to deal with the nonlocal boundary conditions by using quartic splines [38], and the method is very easier than the numerical integration trapezoidal rule [32].

Secondly, for different steps $h = k = 1/20$, $h = k = 1/40$ and $h = 1/40$, $k = 1/60$, the example is solved by using HPSO. The maximum absolutes errors of the numerical solutions are calculated and compared with those results obtained by the method in [33], see Table 3. In [33], Caglar et al. developed a numerical method by using third degree B-splines functions, but the accuracy of the third degree B-splines approximation method is only second order, and the approximation order at the end points is third order.

Table 2
Relative errors at various step.

Step	0.0500	0.0250	0.0100	0.0050	0.0025	0.0010
[32]	9.6E−3	2.5E−3	3.9E−4	9.6E−5	2.5E−5	4.3E−6
[42]	9.1E−3	2.3E−3	3.8E−4	9.4E−5	2.3E−5	4.1E−6
[43]	9.9E−2	3.0E−2	4.9E−3	1.2E−3	3.1E−4	5.0E−5
[44]	9.4E−2	2.4E−2	4.1E−3	1.0E−3	2.5E−4	4.0E−5
[45]	9.8E−2	3.7E−2	6.1E−3	1.5E−3	3.5E−4	6.0E−5
CGM	1.2E−4	3.0E−5	4.4E−6	1.1E−6	2.8E−7	4.5E−8
PSO	4.3E−2	8.9E−3	5.1E−3	2.1E−3	3.3E−4	2.6E−5
HPSO	1.0E−4	1.3E−5	2.1E−6	1.0E−6	1.4E−7	2.0E−8

6.4. Performance of PHPSO

Parallelization is a very important strategy for reducing the computing time for corresponding sequential algorithms. In this experiment, we evaluate our algorithm on multi-core CPU and GPU systems to accelerate the computing speed of HPSO.

6.4.1. Experimental platform setup

In this paper, apart from the proposed GPU implementation, we also implement the algorithm on multi-core CPU system with shared memory architectures (using OpenMP as programming model), so as to compare the performance of two GPU versions with one multi-core CPU version.

A series of experiments are carried out on a dual-processor eight-core 2.3 GHz AMD Opteron 6134 machine with 8 GB main memory. Two different NVIDIA graphics cards GTX 465 and Tesla C2050 are used to check the scalability of our approach as follows:

- The GTX 465 has 11 multiprocessors, 352 CUDA cores, 48 kB of shared memory per block, 607 MHz processor clock, 1 GB GDDR5 RAM, 102.6 GB/s memory bandwidth, and 802 MHz memory clock.
- The Tesla C2050 has 14 multiprocessors, 448 CUDA cores, 48 kB of shared memory per block, 1.15 GHz processor clock, 3 GB GDDR5 RAM, 144 GB/s memory bandwidth, and 1.5 GHz memory clock.

For these two cards, the maximum number of resident blocks per multiprocessor is 8, the maximum number of threads per block is 1024, the maximum number of resident threads per multiprocessor is 1536, and the total number of registers available per block is 32,768.

In software, the testing system is built on top of the Linux (Ubuntu 10.10) operating system, the NVIDIA CUDA Driver version 4.2, and GCC version 4.4.5.

6.4.2. Result analyses of PHPSO

For each test, we first run the sequential algorithm on a single core of the CPU and obtain the computational time. Secondly, we run the parallel algorithm on 16 cores of the CPU with 16 concurrent threads. Finally, we run the parallel algorithm on the GTX 465 GPU and Tesla C2050 GPU, respectively. To avoid system jitter,

Table 3
The Maximum absolute errors at various step h and k.

Steps	[33]	CGM	PSO	HPSO
$h = k = 1/20$	6.2E−3	2.8E−4	5.3E−2	1.5E−4
$h = k = 1/40$	1.6E−3	6.2E−5	1.4E−2	3.4E−5
$h = 1/40,\ k = 1/60$	9.6E−4	3.6E−5	3.3E−3	1.3E−5

Table 4
The speedup of PCGM with GTX465.

Step	CGM (ms)	PCGM (ms)	Speedup	CGM (std)	PCGM (std)
0.0500	3701.24	1075.94	3.44	8.2E−01	7.8E−01
0.0250	7809.62	1743.22	4.48	3.8E−01	3.4E−01
0.0100	18899.27	3405.27	5.55	4.8E−02	5.9E−02
0.0050	36853.58	4600.95	8.01	2.5E−02	5.4E−02
0.0025	76655.45	8172.22	9.38	3.1E−02	2.3E−02
0.0010	200070.71	17262.36	11.59	1.2E−02	1.9E−02

Table 5
The speedup of PPSO with GTX465.

Step	PSO (ms)	PPSO (ms)	Speedup	PSO (std)	PPSO (std)
0.0500	2635.14	1365.36	1.93	8.6E−02	4.5E−04
0.0250	6547.86	2762.81	2.37	8.5E−03	7.8E−05
0.0100	15484.62	4931.41	3.14	9.2E−03	8.6E−05
0.0050	30018.98	6791.62	4.42	5.7E−03	3.2E−05
0.0025	68975.27	11515.07	5.99	6.3E−04	2.9E−05
0.0010	162562.02	22640.95	7.18	4.0E−04	1.7E−05

Table 6
The speedup of PHPSO with GTX465.

Step	HPSO (ms)	PHPSO (ms)	Speedup	HPSO (std)	PHPSO (std)
0.0500	3845.23	852.60	4.51	8.9E−03	5.5E−08
0.0250	7925.46	1096.19	7.23	5.3E−04	3.2E−08
0.0100	19965.25	2112.72	9.45	7.7E−04	5.1E−09
0.0050	38014.85	2830.59	13.43	7.1E−05	4.5E−09
0.0025	78987.26	4766.88	16.57	5.9E−05	3.2E−09
0.0010	230415.22	10692.12	21.55	4.4E−06	2.2E−10

Table 7
The speedup of PCGM with Tesla C2050.

Step	CGM (ms)	PCGM (ms)	Speedup	CGM (std)	PCGM (std)
0.0500	3701.24	1025.27	3.61	6.7E−01	5.6E−01
0.0250	7809.62	1701.44	4.59	2.2E−01	2.7E−01
0.0100	18899.27	3247.30	5.82	7.1E−02	6.3E−02
0.0050	36853.58	4477.96	8.23	5.5E−02	3.9E−02
0.0025	76655.45	8035.16	9.54	3.9E−02	2.4E−02
0.0010	200070.71	16955.15	11.80	2.8E−02	1.4E−02

Table 8
The speedup of PPSO with Tesla C2050.

Step	PSO (ms)	PPSO (ms)	Speedup	PSO (std)	PPSO (std)
0.0500	2635.14	1273.01	2.07	5.0E−02	2.2E−04
0.0250	6547.86	2305.58	2.84	8.6E−02	8.4E−05
0.0100	15484.62	3785.97	4.09	7.2E−03	6.7E−05
0.0050	30018.98	5528.36	5.43	5.4E−03	4.6E−05
0.0025	68975.27	10828.14	6.37	3.7E−03	3.6E−05
0.0010	162562.02	21617.29	7.52	2.4E−04	2.0E−05

Table 9
The speedup of PHPSO with Tesla C2050.

Step	HPSO (ms)	PHPSO (ms)	Speedup	HPSO (std)	PHPSO (std)
0.0500	3845.23	767.51	5.01	6.8E−04	4.4E−08
0.0250	7925.46	1045.58	7.58	5.4E−05	2.6E−08
0.0100	19965.25	1959.30	10.19	3.4E−05	3.4E−08
0.0050	38014.85	2621.71	14.50	5.9E−05	5.9E−09
0.0025	78987.26	4400.40	17.95	2.8E−06	2.8E−09
0.0010	230415.22	9970.37	23.11	2.0E−07	2.0E−10

each experiment is repeated 30 times and the average value is considered. In terms of execution time, we only report the running time for the three stages of the algorithm, and do not report the time for other operations. Remarkably, the time associated with the data transfer between CPU and GPU is included in the execution time for the GPU implementation. Traditionally, the speedup is defined as sequential execution time over parallel execution time.

Moreover, to achieve the most effective and robust performance for the GPU implementation, we should select the optimum numbers of threads per block (TPB) to keep the GPU multiprocessors as active as possible. Therefore, we first conduct our preliminary speedup performance tests on the Tesla C2050 GPU using six different sizes of TPB (respectively, 32, 64, 128, 256, 512, and 1024). The NVIDIA GPUs used here only allow the CUDA kernel to launch a maximum of 1024 TPB.

The computational experiments focus on two aspects of performance: speedup comparison and the robustness comparison. Then results are reported below. The aim of the first set of experiments is to evaluate which part of the proposed PHPSO algorithm improves over its corresponding CPU implementation. A real case of heat conduction equation is selected to process this investigation. Tables 6 and 9 present the comparison of computation times between the PHPSO and the CPU implementation of the same algorithm. Currently, both implementations have used single precision floating numbers, as it meets the precision requirements for the objective function and provides much better performance in speed than double precision floating numbers on a current GPU [46].

We can see that: the average speedup of PHPSO is 12.12 over HPSO with GPU GTX465 in Table 6; the average speedup of PHPSO is 13.06 over HPSO with GPU GTX465 in Table 9. These data indicate that the speedup of PHPSO is ideal for the one-dimensional heat conduction equation. In all the tables, the "std" denotes the standard deviation of statistical data. The standard deviation of PHPSO on GPU is much smaller than that of HPSO. It indicates that the robustness of PHPSO is stronger than HPSO. The random number generated by GPU is better than that of CPU, consequently, the quality of PHPSO is better than that of HPSO. We can conclude that the parallel implementation of HPSO is very significant.

After comparing the speedups of three parallel methods between Tables 4–9, we can find that the speedups of three methods on GTX465 is a little smaller than those of on Tesla C2050. The average speedup of PCGM at various steps (spatial step is equal to time step) with GTX465 and Tesla C2050 is respectively 7.08, 7.27; The average speedup of PPSO at various steps (spatial step is equal to time step) with GTX465 and Tesla C2050 is respectively 4.17,

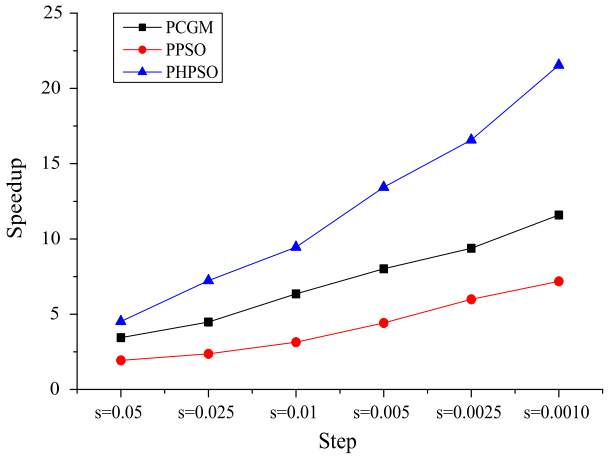

Fig. 5. The speedups of three methods on GTX465.

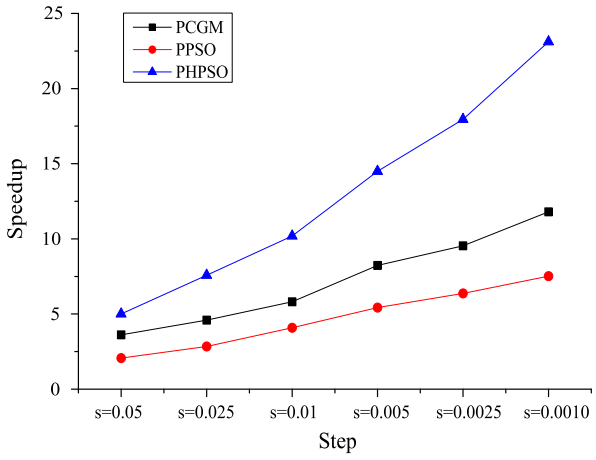

Fig. 6. The speedups of three methods on Tesla C2050.

4.72. However, the average speedup of PHPSO at various step (spatial step is equal to time step) with GTX465 and Tesla C2050 is respectively 12.12, 13.06. Obviously, Figs. 5 and 6 show that the speedup of PHPSO is much higher than that of PCGM and PPSO. Consequently, the parallel implementation of CGM-only or PSO-only is not so ideal and it is necessary for the parallel implementation to hybridize PSO and CGM.

We can see that the standard deviation of PHPSO at various steps is smaller than that of PPSO and much smaller than that of PCGM. Consequently, the robustness of PHPSO is stronger than that of PCGM and PPSO. It is completely obvious that the hybrid implementation of CGM and PSO (PHPSO) is very significant.

7. Conclusion

One-dimensional heat conduction equation is highly common in the science research field and engineering application. We adopt quartic spline and difference scheme to transform the heat conduction equation into a system of linear equations. In order to employ PSO and HPSO to solve it, a system of linear equations is successfully transformed into an unconstrained optimization problem. A parallel hybrid PSO (PHPSO) algorithm is designed to solve a real case of one-dimensional heat conduction equation. We therefore implement the PHPSO in CUDA for use on GPUs and verify its performance. The relative error and abstract error of calculation results for the PHPSO is the same as the HPSO and the standard deviation of PHPSO is much smaller than the PHPSO. The speedups of PHPSO on two different GPUs are up to an average of 12.12 and 13.06 times faster, respectively, for different time and spatial steps.

The programs of PPSO, PCGM and PHPSO are implemented on two various GPUs, the speedups and standard deviations of three algorithms are displayed in this study. Comparison of three parallel algorithms shows that the PHPSO is competitive in terms of speedup and standard deviation. The results also show that using PHPSO to solve the one-dimensional heat conduction equation can outperform two parallel algorithms as well as HPSO itself. Thus, the PHPSO is an efficient and effective approach towards the heat conduction equation, as it is shown to be with strong robustness and high speedup.

Acknowledgements

This paper is partially funded by the National Natural Science Foundation of China (Grant Nos. 61070057, 61133005, 61370095, 61103047, 61202109), the Project Supported by Scientific Research Fund of Hunan Provincial Education Department (Nos. 13C333, 08D092), the Project Supported by the Science and Technology Research Foundation of Hunan Province (Grant Nos. 2013GK3082, 2014GK3043)

References

[1] Satake S, Yoshimori H, Suzuki T. Optimizations of a GPU accelerated heat conduction equation by a programming of CUDA Fortran from an analysis of a PTX file. Comput Phys Commun 2012;183(11):2376–85.
[2] Shi L, Chen H, Sun J, Li K. vCUDA: GPU-accelerated high-performance computing in virtual machines. IEEE Trans Comput 2012;61(6):804–16.
[3] Chen H, Shi L, Sun J, Li K, He L. A fast RPC system for virtual machines. IEEE Trans Parallel Distrib Syst 2013;24(7):1267–76.
[4] Liu X, Li M, Li S, Peng S, Liao X, Lu X, et al. IMGPU: GPU accelerated influence maximization in large-scale social networks. IEEE Trans Parallel Distrib Syst 2014;25(1):136–45.
[5] Rustico E, Bilotta G, Herault A, Del Negro C, Gallo G. Advances in multi-GPU smoothed particle hydrodynamics simulations. IEEE Trans Parallel Distrib Syst 2014;25(1):43–52.
[6] Nelson B, Kirby RM, Haimes R. GPU-based volume visualization from high-order finite element fields. IEEE Trans Visual Comput Graph 2014;20(1):70–83.
[7] Mei S, He M, Shen Z. Optimizing hopfield neural network for spectral mixture unmixing on GPU platform. IEEE Geosci Remote Sens Lett 2014;11(4):818–22.
[8] Chitchian M, Simonetto A, van Amesfoort A, Keviczky T. Distributed computation particle filters on GPU architectures for real-time control applications. IEEE Trans Control Syst Technol 2013;21(6):2224–38.
[9] Rodriguez-Losada D, San Segundo P, Hernando M, de la Puente P, Valero-Gomez A. GPU-mapping: robotic map building with graphical multiprocessors. IEEE Robot Autom Mag 2013;20(2):40–51.
[10] Richter C, Schops S, Clemens M. GPU acceleration of finite difference schemes used in coupled electromagnetic/thermal field simulations. IEEE Trans Magn 2013;49(5):1649–52.
[11] Fu Z, Lewis TJ, Kirby RM, Whitaker RT. Architecting the finite element method pipeline for the GPU. J Comput Appl Math 2014;257:195–211.
[12] Orchard G, Martin J, Vogelstein R, Etienne-Cummings R. Fast neuromimetic object recognition using FPGA outperforms GPU implementations. IEEE Trans Neural Networks Learn Syst 2013;24(8):1239–52.
[13] Feng Z, Li P. Fast thermal analysis on GPU for 3D ICs with integrated microchannel cooling. IEEE Trans Very Large Scale Integr (VLSI) Syst 2013;21(8):1526–39.
[14] Westphal E, Singh S, Huang C-C, Gompper G, Winkler R. Multiparticle collision dynamics: GPU accelerated particle-based mesoscale hydrodynamic simulations. Comput Phys Commun 2014;185(2):495–503.
[15] Tuttafesta M, DAngola A, Laricchiuta A, Minelli P, Capitelli M, Colonna G. GPU and multi-core based reaction ensemble Monte Carlo method for non-ideal thermodynamic systems. Comput Phys Commun 2014;185(2):540–9.
[16] Jie L, Li K, Shi L, Liu R, Mei J. Accelerating solidification process simulation for large-sized system of liquid metal atoms using GPU with CUDA. J Comput Phys 2014;257, Part A:521–35.
[17] Ye Y, Li K. Entropic lattice Boltzmann method based high Reynolds number flow simulation using CUDA on GPU. Comput Fluids 2013;88:241–9.
[18] Leskinen J, Priaux J. Distributed evolutionary optimization using Nash games and GPUs-applications to CFD design problems. Comput Fluids 2013;80:190–201.
[19] Hashimoto T, Tanno I, Tanaka Y, Morinishi K, Satofuka N. Simulation of doubly periodic shear layers using kinetically reduced local Navier–Stokes equations on a GPU. Comput Fluids 2013;88:715–8.
[20] Hu Q, Gumerov NA, Duraiswami R. GPU accelerated fast multipole methods for vortex particle simulation. Comput Fluids 2013;88:857–65.
[21] Qi H-T, Xu H-Y, Guo X-W. The Cattaneo-type time fractional heat conduction equation for laser heating. Comput Math Appl 2013;66(5):824–31.
[22] Hosseini S, Shahmorad S, Masoumi H. Extension of the operational tau method for solving 1-d nonlinear transient heat conduction equations. J King Saud Univ – Sci 2013;25(4):283–8.
[23] Yılmazer A, Kocar C. Exact solution of the heat conduction equation in eccentric spherical annuli. Int J Therm Sci 2013;68:158–72.
[24] Liu J, Zheng Z. A dimension by dimension splitting immersed interface method for heat conduction equation with interfaces. J Comput Appl Math 2014;261:221–31.
[25] Liu L-B, Liu H-W, Chen Y. Polynomial spline approach for solving second-order boundary-value problems with Neumann conditions. Appl Math Comput 2011;217(16):6872–82.
[26] Liu L-B, Liu H-W. A new fourth-order difference scheme for solving an n-carrier system with Neumann boundary conditions. Int J Comput Math 2011;88(16):3553–64.
[27] Cao H-H, Liu L-B, Zhang Y, Fu S-m. A fourth-order method of the convection–diffusion equations with Neumann boundary conditions. Appl Math Comput 2011;217(22):9133–41.
[28] Bougoffa L, Rach RC, Wazwaz A-M. Solving nonlocal initial-boundary value problems for the Lotkac–von Foerster model. Appl Math Comput 2013;225:7–15.
[29] Bougoffa L, Rach RC. Solving nonlocal initial-boundary value problems for linear and nonlinear parabolic and hyperbolic partial differential equations by the Adomian decomposition method. Appl Math Comput 2013;225:50–61.

[30] Jankowski T. Nonnegative solutions to nonlocal boundary value problems for systems of second-order differential equations dependent on the first-order derivatives. Nonlinear Anal: Theory Methods Appl 2013;87:83–101.
[31] Asanova AT, Dzhumabaev DS. Well-posedness of nonlocal boundary value problems with integral condition for the system of hyperbolic equations. J Math Anal Appl 2013;402(1):167–78.
[32] Dehghan M. The one-dimensional heat equation subject to a boundary integral specification. Chaos Solitons Fract 2007;32(2):661–75.
[33] Caglar H, Ozer M, caglar N. The numerical solution of the one-dimensional heat equation by using third degree b-spline functions. Chaos Solitons Fract 2008;38(4):1197–201.
[34] Liu L-B, Liu H-W. Quartic spline methods for solving one-dimensional telegraphic equations. Appl Math Comput 2010;216(3):951–8.
[35] Tsourkas P, Rubinsky B. A parallel genetic algorithm for heat conduction problems. Numer Heat Transfer Part B 2005;47(2):97–110.
[36] Gosselin L, Tye-Gingras M, Mathieu-Potvin F. Review of utilization of genetic algorithms in heat transfer problems. Int J Heat Mass Transfer 2009;52(9-10):2169–88.
[37] He L, Zou D, Zhang Z, Chen C, Jin H, Jarvis SA. Developing resource consolidation frameworks for moldable virtualmachines in clouds. Future Gener Comp Syst 2014;32:69–81.
[38] Liu H-W, Liu L-B, Chen Y. A semi-discretization method based on quartic splines for solving one-space-dimensional hyperbolic equations. Appl Math Comput 2009;210(2):508–14.
[39] Kennedy J, Eberhart R. Particle swarm optimization. Proceedings of IEEE international conference on neural networks, vol. 4. Piscataway, NJ, USA: IEEE Press; 1995. p. 1942–8.
[40] Ouyang A, Tang Z, Li K, Sallam A, Sha E. Estimating parameters of Muskingum model using an adaptive hybrid PSO algorithm. Int J Pattern Recognit Artif Intell 2014;28(1):1–29.
[41] Mussi L, Daolio F, Cagnoni S. Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture. Inform Sci 2011;181(20):4642–57.
[42] Cannon JR, Esteva SP, Hoek JVD. A Galerkin procedure for the diffusion equation subject to the specification of mass. SIAM J Numer Anal 1987;24(3):499–515.
[43] Cannon JR, Matheson AL. A numerical procedure for diffusion subject to the specification of mass. Int J Eng Sci 1993;31(3):347–55.
[44] Ewing RE, Lin T. A class of parameter estimation techniques for fluid flow in porous media. Adv Water Resour 1991;14(2):89–97.
[45] Wang Y, Liu L, Wu Y. Positive solutions for singular semipositone boundary value problems on infinite intervals. Appl Math Comput 2014;227:256–73.
[46] Zhu W. Nonlinear optimization with a massively parallel evolution strategy–pattern search algorithm on graphics hardware. Appl Soft Comput 2011;11(2):1770–81.