GPU-based Parallel Particle Swarm Optimization

You Zhou, and Ying Tan, *Senior Member, IEEE*

***Abstract*— A novel parallel approach to run standard particle swarm optimization (SPSO) on Graphic Processing Unit (GPU) is presented in this paper. By using the general-purpose computing ability of GPU and based on the software platform of Compute Unified Device Architecture (CUDA) from NVIDIA, SPSO can be executed in parallel on GPU. Experiments are conducted by running SPSO both on GPU and CPU, respectively, to optimize four benchmark test functions. The running time of the SPSO based on GPU (GPU-SPSO) is greatly shortened compared to that of the SPSO on CPU (CPU-SPSO). Running speed of GPU-SPSO can be more than 11 times as fast as that of CPU-SPSO, with the same performance. compared to CPU-SPSO, GPU-SPSO shows special speed advantages on large swarm population applications and hign dimensional problems, which can be widely used in real optimizing problems.**

I. Introduction

Particle swarm optimization(PSO), developed by Eberhart and Kennedy in 1995, is a stochastic global optimization technique inspired by social behavior of bird flocking or fish schooling [1]. In the PSO, each particle in the swarm adjusts its position in the search space based on the best position it has found so far as well as the position of the known best-fit particle of the entire swarm, and finally converges to the global best point of the whole search space.

Compared to other swarm based algorithms such as genetic algorithm and ant colony algorithm, PSO has the advantage of easy implementation, while maintaining strong abilities of convergence and global search. In recent years, PSO has been used increasingly as an effective technique for solving complex and difficult optimization problems in practice. PSO has been successfully applied to problems such as function optimization, artificial neural network training, fuzzy system control, blind source separation, machine learning and so on.

In spite of those advantages, PSO still needs a long time to find solutions for large scale problems, such as problems with large dimensions and problems which need a large swarm population for searching in the solution space. The main reason for this is that the optimizing process of PSO requires a large number of fitness evaluations, which are usually done in a sequential way on CPU, so the computation task can be very heavy and thus running speed of PSO may be quite slow.

In recent years, Graphics Processing Unit (GPU) which has traditionally been a graphics-centric workshop, has shifted its attention to the non-graphics and general-purpose computing applications. Because of its parallel computing mechanism and fast float-point operation, GPU has shown great advantages in scientific computing fields, and achieved many successful applications.

In order to perform general-purpose computing on GPU more easily and conveniently, some platforms have been developed, such as BrookGPU (Stanford University) [2], CUDA (Compute Unified Device Architecture, NVIDIA Corporation) [3]. These platforms have greatly simplified programming on GPU.

In this paper, we present a novel method to run PSO on GPU in parallel, based on CUDA, which is a new but powerful platform for programming on GPU. With a good optimization performance, the PSO implemented on GPU can enlarge the swarm population and problem dimension sizes, speed up its running greatly and provide users with a feasible solution for complex optimizing problems in reasonable time. As GPU chips can be found in any ordinary PC currently, more and more people will be able to solve huge problems in real-world applications by this parallel algorithm.

The paper is organized as follows. In Section II, the related work is presented in details. In Section III, we briefly introduce backgrounds of GPU based computing. Algorithmic implementations of our proposed approach are elaborated in Section IV. A number of experiments are done on four benchmark functions and analysis of results are reported in Section V. Finally, we conclude the paper and give our future works in Section VI.

II. Related Work

A. Traditional Particle Swarm Optimization

For the sake of simplicity, we call the particle swarm optimization algorithm presented by Eberhart and Kennedy in 1995 [1] as traditional particle swarm optimization (TPSO). In TPSO, each solution of the optimization problem is called a particle in the search space. The search of the problem space is done by a swarm with a specific number of particles. During each of the iteration, the position and velocity of every particle are updated according to its current best position ($P_{pBd}(t)$) and the best position of the entire swarm ($P_{gBd}(t)$). The position and velocity updating in TPSO can be formulated as follows:

$$V_{id}(t+1) = wV_{id}(t) + c_1 r_1 (P_{pBd}(t) - X_{id}(t)) + c_2 r_2 (P_{gBd}(t) - X_{id}(t)) \quad (1)$$

$$X_{id}(t+1) = X_{id}(t) + V_{id}(t) \quad (2)$$

where $i = 1, 2, ...N$, N is the number of particles in the swarm namely the population. $d = 1, 2, ...D$, D is the dimension of solution space. In Equation (1) and (2), the

Y. Zhou and Y. Tan (corresponding author) are with Key Laboratory of Machine Perception and Intelligence (Peking University), Ministry of Education, and with Department of Machine Intelligence, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China (Phone: +86-10-62767611, E-mail: ytan@pku.edu.cn.)

978-1-4244-2959-2/09/$25.00 ©

learning factors c_1 and c_2 are nonnegative constants, r_1 and r_2 are random numbers uniformly distributed in the interval $[0, 1]$, $V_{id} \in [-V_{max}, V_{max}]$, where V_{max} is a designated maximum velocity which is a constant preset according to the objective optimization function. If the velocity on one dimension exceeds the maximum, it will be set to V_{max}. This parameter controls the convergence rate of the PSO and can prevent the method from growing too fast. The parameter w is the inertia weight used to balance the global and local search abilities, which is a constant in the interval $[0, 1]$.

B. Standard Particle Swarm Optimization

In these two decades, many researchers have taken great effort to improve the performance of TPSO by exploring the concepts, issues, and applications of the algorithm, and many variants have also been developed. In spite of this attention, there has as yet been no standard definition representing exactly what is involved in modern implementations of the technique.

In 2007, Daniel Bratton and James Kennedy designed a Standard Particle Swarm Optimization (SPSO) which is a straightforward extension of the original algorithm while taking into account more recent developments that can be expected to improve performance on standard measures [4]. This standard algorithm is intended for use both as a baseline for performance testing of improvements to the technique, as well as to represent PSO to the wider optimization community.

SPSO is different from TPSO mainly in the following aspects [4]:

1) Swarm Communication Topology: TPSO uses a global topology showed in Fig. 1(a). In this topology, the best particle, which is responsible for the velocity updating of all the particles, is chosen from the whole swarm population. While in SPSO there is no global best, every particle only uses a local best particle for velocity updating, which is chosen from its left,right neighbors and itself. We call this a local topology, as shown in Fig. 1(b). (Assuming that the swarm has a population of 12).

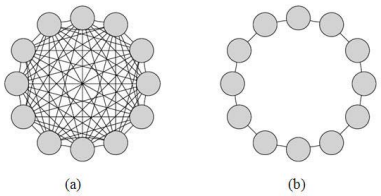

Fig. 1. TPSO and SPSO Topologies

2) Inertia Weight and Constriction: In TPSO, an inertia weight parameter was designed to adjust the influence of the previous particle velocities on the optimization process. By adjusting the value of w, the swarm has a greater tendency to eventually constrict itself down to the area containing the best fitness and explore that area in detail. Similar to the parameter w, SPSO introduced a new parameter χ known as the constriction factor, which is derived from the existing constants in the velocity update equation:

$$\chi = \frac{2}{|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}|}$$
$$\varphi = c_1 + c_2$$

and the velocity updating formula in SPSO is:

$$V_{id}(t+1) = \chi(V_{id}(t) + c_1 r_1(P_{pBd}(t) - X_{id}(t)) + c_2 r_2(P_{gBd}(t) - X_{id}(t))) \quad (3)$$

where P_{gBd} is no longer global best but the local best.

Statistical tests have shown that compared to TPSO, SPSO can return better results, while retaining the simplicity of TPSO. The introduction of SPSO can give researchers a common grounding to work from. SPSO can be used as a means of comparison for future developments and improvements of PSO, and thus prevent unnecessary effort being expended on "reinventing the wheel" on rigorously tested enhancements that are being used at the forefront of the field.

III. Introduction to GPU Based Computing

GPU has already been successfully used in some computation fields such as computer vision problems [5], Voronoi diagrams [6] and neural network computation [7] and so on. GPU is entering the main stream of computing [8].

A. Advantages of GPU on Computing

GPU was at first designed especially for the purpose of image and graphic processing on computers, where compute-intensive and highly parallel computing is required. Compared to CPU, GPU shows many advantages [3]. Firstly, GPU computes faster than CPU. GPU devotes more transistors to data processing rather than data caching and flow control, which enables it to do much more float-point operations per second than CPU. Secondly, GPU is more suitable for data-parallel computations. It is especially well-suited to solve problems that can be expressed as data-parallel computations with high arithmetic intensity - the ratio of arithmetic operations to memory operations.

B. Programming Model for GPU

The programming model for GPU is illustrated by Fig. 2. A shader program operates on a single input element stored in the input registers, then it writes the execution result into the output registers. This process is done in parallel by applying the same operations to all the data.

C. Compute Unified Device Architecture (CUDA)

NVIDIA CUDA technology is a C language environment that enables programmers and developers to write software to solve complex computational problems by tapping into the many-core parallel processing power of GPUs. It is a new hardware and software architecture for issuing and managing computations on the GPU as a data-parallel computing device without the need of mapping them to a graphics API.

Some applications have already been developed based on CUDA, for example, matrix multiplication, parallel prefix

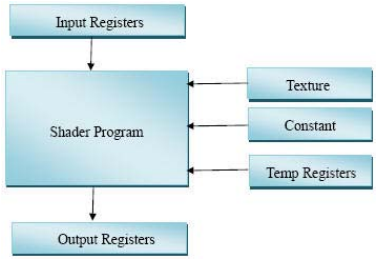

Fig. 2. Programming Model for GPU

sum of large arrays, image denoising, sobel edge detection filter and so on. In this paper, we intend to implement SPSO on GPU in parallel to accelerate the running speed of it.

For details about programming through CUDA, interested readers can refer to the website of NVIDIA CUDA ZONE.

The core concepts in programming through CUDA are **thread batching** and **memory model** [9].

1) Thread Batching: When programming through CUDA, the GPU is viewed as a compute device capable of executing a very high number of threads in parallel. A function in CUDA is called as a Kernel, which is executed by a batch of threads. The batch of threads is organized as a grid of thread blocks. A thread block is a batch of threads that can cooperate together by efficiently sharing data through some fast shared memory and synchronizing their execution to coordinate memory accesses.

Each thread is identified by its thread ID. To help with complex addressing based on the thread ID, an application can also specify a block as a 2 or 3-dimensional array of arbitrary size and identify each thread using a 2 or 3-component index instead. For a two dimensional block of size $D_x \times D_y$, the thread ID of index (x,y) is $y * D_x + x$. The thread batching can be illustrated by Fig. 3.

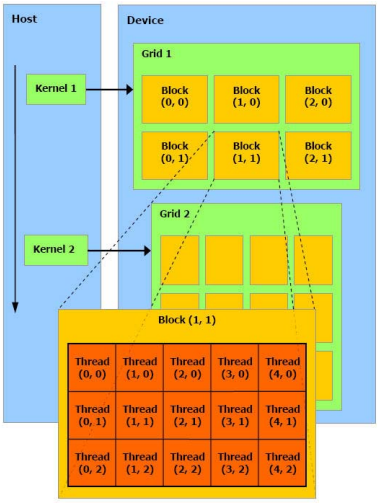

Fig. 3. Thread Batching of a Kernel in CUDA

2) Memory Model: The memory model of CUDA is tightly related to its thread bathing mechanism. There are several kinds of memory spaces on the device:

- Read-write per-thread registers
- Read-write per-thread local memory
- Read-write per-block shared memory
- Read-write per-grid global memory
- Read-only per-grid constant memory
- Read-only per-grid texture memory

The memory model can be illustrated by Fig. 4. Registers and local memory can only be accessed by threads, the shared memory is only accessible within a block, and global memory is available to all the threads in a grid. In this paper, we mainly use the shared memory and global memory for our implementation.

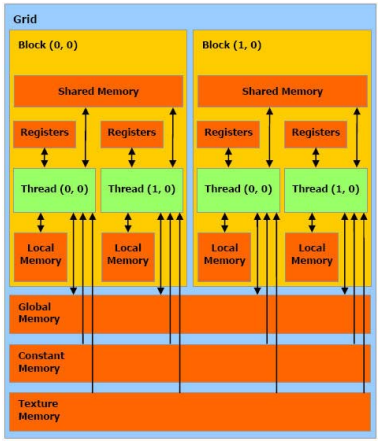

Fig. 4. Memory Model of CUDA

IV. Implementation of SPSO on GPU

SPSO is as simple as TPSO, but with a better performance. So we would implement SPSO rather than TPSO on GPU. Our purpose is to accelerate the running speed of SPSO on GPU (GPU-SPSO), during the search for the global best in an optimization problem. Meanwhile, performances of GPU-SPSO should not be deteriorated. Furthermore, by making full use of the parallel computing ability of GPU, we expect GPU-SPSO can solve optimization problems with high dimension and large swarm population.

A. Data Organization

In this paper, position and velocity information of all the particles is stored on the global memory of GPU chips. As the global memory only allows the allocation of one dimensional arrays, so only one-dimensional arrays are used here for storing data, including the position, velocity and fitness values of all the particles.

Let us assume that the dimension of the problem is D, and the swarm population is N. An array X of length $D * N$ is used here to represent this swarm by storing all the position values. But the array should be logically seen as a two-dimensional array Y. An element with the index of (i, j) in Y corresponds to the element in X with the index

of ($i*N+j$). This address mapping method can be formulized as:

$$Y(i,j) = X(i*N+j) \tag{4}$$

where the element $Y(i,j)$ stands for the datum for the i-th dimension of the j-th particle in the swarm.

B. Variable Definition and Initialization

Suppose the fitness value function of the problem is $f(X)$ in the domain $[-r,r]$. The dimension of the problem is D and swarm population is N. The variable definitions are given as follows: (each array is one-dimensional)

- Particle position array: X
- Particle velocity array: Y
- Personal best position: PX
- Local best position: GX
- Fitness value of particles: F
- Personal best fitness value: PF
- Local best fitness value: GF

where the sizes of X,Y,PX and GX are all $D*N$; the sizes of F,PF,and GF are all N.

C. Random Number Generation

During the process of optimization, SPSO needs lots of random numbers for velocity updating. The absence of high precision integer arithmetic in current generation GPUs makes random numbers generating on GPU very tricky though it is still possible [10]. In order to focus on the implementation of SPSO, we would rather generate random numbers on CPU and transport them to GPU. However the data transportation between GPU and CPU is quite time consuming. If we generate random numbers on CPU and transfer them to GPU during each iteration of SPSO, it will greatly slow down the algorithm's running speed due to mountains of data to be transported. So data transportation between CPU and GPU should be avoided as much as possible.

We solve this problem like this: M ($M >> D*N$) random numbers are generated on CPU before running SPSO. Then they are transported to GPU once for ado and stored in an array R on the global memory. When the velocity updating is carried through, we just pass two random integer numbers P_1,$P_2 \in$ [0,M-$D*N$] from CPU to GPU, then 2*D*N numbers can be drawn from array R starting at $R(P_1)$ and $R(P_2)$, respectively, instead of transporting $2*D*N$ numbers from CPU to GPU. The running speed can be obviously improved by using this technique.

D. Algorithmatic Flow for GPU-SPSO

The algorithm flows for GPU-SPSO is illustrated by Algorithm 1. Here $Iter$ stands for the maximum number of iterations that GPU-SPSO runs, which serves as the stop condition for the optimization process.

Algorithm 1 Algorithmatic Flow for GPU-SPSO

Initialize the positions and velocities of all particles.
Transfer data from CPU to GPU.

// sub-processes in "for" are done in parallel
for i=1 to $Iter$ **do**
  Compute fitness values of all particles
  Update $pBest$ of each particle
  Update $gBest$ of each particle
  Update velocity and position of each particle
end for
Transfer data back to CPU and output.

E. Parallelization Design

The difference between a CPU function and a GPU **kernel** is that execution of the kernel should be parallelized. So we must design the parallelization methods for all the sub-processes of optimizing by SPSO.

1) Compute Fitness Values of Particles: The computing of fitness values is the most important computation task in the whole search process, where high density of arithmetical computation is required. It should be carefully designed for parallelization so as to improve the overall performance (we consider mainly running speed) of GPU-SPSO.

The algorithm for fitness values computing of all the particles is shown in Algorithm 2.

Algorithm 2 Compute Fitness Values

Initialize, set the *'block size'* and *'grid size'*, with the number of threads equaling to the number of particles N.

for each dimension i **do**
  Map all threads to the N position values one-to-one
  Load N data from global to shared memory
  Apply arithmetical operations to all N data in parallel
  Store the result of dimension i with $f(X_i)$
end for

Combine $f(X_i)$ ($i = 1, 2...D$) to get the final fitness values $f(X)$ of all particles, store them in array F.

From Algorithm 2, we can see that the iteration is only applied to dimension index $i = 1, 2..., D$, while on CPU, it should also be applied to the particle index $j = 1, 2, ...N$. The reason is that the arithmetical operation to all the N data in dimension i is done in parallel (synchronously) on GPU.

Mapping all the threads to the N data in a 1-D array should follow two steps:

- Set the block size to $S_1 \times S_2$ and grid size $T_1 \times T_2$. So the total number of threads in the grid is $S_1*S_2*T_1*T_2$. It must be guaranteed that $S_1 * S_2 * T_1 * T_2$=N, only in this case can all the data of N particles be loaded and processed synchronously.
- Assuming that the thread with the index (T_x, T_y) in the block whose index is (B_x, B_y), is mapped to the Ith

datum in a 1-D array, then the relationship between the indexes and I is:

$$I = (B_y * T_2 + B_x) * S_1 * S_2 + T_y * S_2 + T_x \quad (5)$$

In this way, all the threads in a Kernel are mapped to N data one to one. Then an operation to one thread will cause all the N threads to do exactly the same operation synchronously. This is the core mechanism for explaining why GPU can accelerate the computing speed greatly.

2) Update $pBest$ and $gBest$: After the fitness values are updated, each particle may come to a better place than ever before and new local best particles may be found. So $pBest$ and $gBest$ (refer to Equation. 1) must be updated according to the current status of the particle swarm. The updating of $pBest$ (namely PX and PF) can be done by Algorithm 3.

Algorithm 3 Update $pBest$

Map all the threads to N particles one-to-one.
Transfer all the N data from global to shared memory.

//Do operations to thread i ($i = 1, ..., N$) synchronously:
if $F(i)$ is better than $PF(i)$ **then**
  $PF(i)$= $F(i)$
  for each dimension d **do**
    Store the position $X(d * N + i)$ to $PX(d * N + i)$
  end for
end if

The updating of $gBest$ (namely GX and GF) is similar to that of $pBest$. Compare a particle's previous $gBest$ to the current $pBest$ of the right neighbor, left neighbor and its own, respectively, then choose the fittest as the new $gBest$ for that particle.

3) Update Velocity and Position: After the personal best and local best positions of all the particles have been updated, the velocities and positions should also be updated according to Equation. 3 and Equation. 2, respectively, by making use of the new information provided by $pBest$ and $gBest$. This process is done dimension by dimension. In the same dimension d ($d = 1, ..., D$), the velocities of all the particles are updated in parallel, using the same technique mentioned in the previous algorithms. What should be paid special attention to is that two random integers P_1 and P_2 should be provided for fetching random numbers from array R.

V. Experimental Results and Analysis

The experimental platform for this paper is based on Intel Core 2 Duo 2.20GHz CPU, 2.0GRAM, NVIDIA GeForce 8600GT, and Windows XP.

In this paper, performance comparisons between GPU-SPSO and CPU-SPSO is made based on four classical benchmark test functions as listed in TABLE I.

The variables of f_1,f_2,andf_3 are independent but variables of f_4 are dependent, namely there are related variables such as the i-th and $(i+1)$-th variable. The optimal solution of all the four functions are 0.

TABLE I
Benchmark Test Functions

Name	Equation	Bounds
f_1	$\sum_{i=1}^{D} X_i^2$	$(-100, 100)^D$
f_2	$\sum_{i=1}^{D} [x_i^2 - 10 * \cos(2\pi x_i) + 10]$	$(-10, 10)^D$
f_3	$\frac{1}{4000} \sum_{i=1}^{D} X_i^2 - \prod_{i=1}^{D} \cos(x_i/\sqrt{i}) + 1$	$(-600, 600)^D$
f_4	$\sum_{i=1}^{D-1} (100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2)$	$(-10, 10)^D$

SPSO is run both on GPU and CPU in this paper. We call them as GPU-SPSO and CPU-SPSO, respectively. Now we define **Speedup** as the times that GPU-SPSO runs faster than CPU-SPSO.

$$\gamma = \frac{T_{CPU}}{T_{GPU}} \quad (6)$$

where γ is **Speedup**, T_{CPU} and T_{GPU} is, respectively, the time that CPU-SPSO and GPU-SPSO need to optimize a function during a specific number of iterations.

In the following paragraphs, **Iter** stands for the number of iterations that SPSO runs, **D** is dimension, **N** is swarm population; **CPU-Time** and **GPU-Time** stand for the time consumed for running SPSO on CPU and GPU, respectively, with second as unit of time. **CPU-Value** and **GPU-Value** stand for the mean final optimized function values of the 10 runs on CPU and GPU, respectively.

The experimental results and analysis are given as follows.

A. Running Time and Speedup Versus Swarm Population

We run both GPU-SPSO and CPU-SPSO on f_1,f_2 andf_3 for 10 times independently, and the results are shown in TABLE II, III, IV. (D=50, $Iter$=2000)

TABLE II
Results of CPU-SPSO and GPU-SPSO on f_1 (D=50)

N	CPU-Time(s)	GPU-Time(s)	**Speedup**	CPU-Value	GPU-Value
400	47.9893	12.8183	**3.7438**	4.66E-8	4.90E-8
1200	153.4973	33.0714	**4.6414**	3.63E-8	3.21E-8
2000	250.3312	55.1357	**4.5402**	3.02E-8	2.85E-8
2800	362.737	72.775	**4.9843**	2.80E-8	2.93E-8

TABLE III
Results of CPU-SPSO and GPU-SPSO on f_2 (D=50)

N	CPU-Time(s)	GPU-Time(s)	**Speedup**	CPU-Value	GPU-Value
400	86.6102	14.4711	**5.9850**	135	133.4977
1200	261.9971	35.132	**7.4575**	107	109.1144
2000	449.6615	59.6859	**7.5338**	102	106.1681
2800	642.576	79.1055	**8.1230**	96	101.6261

TABLE IV

RESULTS OF CPU-SPSO AND GPU-SPSO ON f_3 (D=50)

N	CPU-Time(s)	GPU-Time(s)	**Speedup**	CPU-Value	GPU-Value
400	96.3025	14.3078	**6.4737**	8.87E-10	5.96E-09
1200	282.2462	34.4804	**8.1857**	5.57E-10	0
2000	481.5713	57.6654	**8.3511**	6.75E-10	0
2800	677.2406	74.8407	**9.0490**	4.02E-10	0

After analyzing the data given in TABLE II, III, IV, we can make some conclusions below.

1) Optimization Performance Comparison: GPU-SPSO uses a random number "pool" for updating velocity instead of instant random numbers generation. It may affect the results to some extent. However, seen from the tables, on all the three functions, GPU-SPSO and CPU-SPSO can give optimal results in the same magnitude, namely the GPU-SPSO can almost reach the same results precision as CPU-SPSO (even better, for example f_3). So we can say that GPU-SPSO is reliable and efficient in total.

2) Population Size Setup: When the population size grows (from 400 to 2800), the optima of a function to be found are in the same magnitude, no obvious precision improvement can be seen. So we can say that a large swarm population is not always necessary. But exceptions may still exist where large swarm population is required in order to obtain better optimization results especially in real-world optimization problems.

3) Running Time: As shown in Fig. 5, The running time of GPU-SPSO and CPU-SPSO is proportional to the swarm population, namely the time increase linearly with swarm population, keeping the other parameters constant.

From f_1, f_2 to f_3, the complexity of computation increases. f_1 contains only square arithmetic, f_2 contains square and cosine arithmetic, while f_3 contains not only square and cosine but also square root arithmetic. In the case of same population size, it takes much more time for CPU-SPSO to optimize the function with more complex arithmetic than the one with less complex arithmetic, during the same number of iterations. However, this is no longer true for GPU-SPSO. Notice that the three lines which stand for time consumed by the GPU-SPSO when optimizing f_1, f_2, $f_3$overlap each other in Fig. 5, namely the time remains almost the same for a GPU-SPSO swarm with a specific population to optimize them, during the same number of iterations. So more complex arithmetic a function has, more speed advantages can GPU-SPSO gain compared to CPU-SPSO when optimizing it.

4) Speedup: As seen from Fig. 6, the speedup of the same function increases with the population size, but it is limited to a specific constant. Furthermore, the line of a function with more complex arithmetics lies above the line of the functions with less complex arithmetics (f_3 above f_2, f_2 above f_1), that is to say the function with more complex arithmetic has a higher speedup.

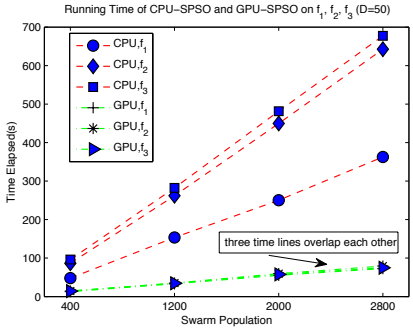

Fig. 5. Running Time and Swarm Population

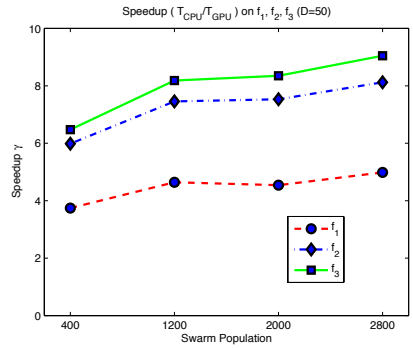

Fig. 6. Speedup and Swarm Population

B. Running Time and Speedup versus Dimension

Now we fix the swarm population to a constant number and vary the dimension. Analysis about the relationship between running time (as well as speedup) and dimension is done here. We run both GPU-SPSO and CPU-SPSO on f_1,f_2 and f_3 for 10 times, and the results are shown in TABLE V, VI, VII (N=400, $Iter$=2000).

TABLE V

RESULTS OF CPU-SPSO AND GPU-SPSO ON f_1 (N=400)

D	CPU-Time(s)	GPU-Time(s)	**Speedup**	CPU-Value	GPU-Value
50	118.4688	31.2999	**3.785**	0	0
100	237.616	60.556	**3.9239**	2.72E-10	2.80E-10
150	356.7932	89.5998	**3.9821**	5.18E-05	5.18E-05
200	483.0046	119.9542	**4.0266**	0.0232	0.0235

From TABLE V, VI, VII, we can conclude that:

1) Running Time: As seen from Fig. 7, the running time of GPU-SPSO and CPU-SPSO increases linearly with dimension, keeping the other parameters constant. Functions (f_2 and f_3) with more complex arithmetic need more time than the function (f_1) with much less complex arithmetic to be optimized by CPU-SPSO, while the time needed is almost the same when optimized by GPU-SPSO, with the same problem dimension. Just as mentioned in Section V-A.3.

TABLE VI

RESULTS OF CPU-SPSO AND GPU-SPSO ON f_2 (N=400)

D	CPU-Time(s)	GPU-Time(s)	**Speedup**	CPU-Value	GPU-Value
50	274.1156	35.444	**7.7338**	122.3175	117.047
100	493.2735	67.8662	**7.2683**	4.45E+02	448.3257
150	753.7683	101.4741	**7.4282**	8.14E+02	8.77E+02
200	996.2754	133.2195	**7.4845**	1.36E+03	1.43E+03

TABLE VII

RESULTS OF CPU-SPSO AND GPU-SPSO ON f_3 (N=400)

D	CPU-Time(s)	GPU-Time(s)	**Speedup**	CPU-Value	GPU-Value
50	242.1754	33.2244	**7.2891**	0	0
100	466.4977	66.8614	**6.9771**	1.14E-11	0
150	724.5183	98.9491	**7.3221**	4.76E-06	2.33E-05
200	1020.6	130.8583	**7.7989**	0.0018	0.0062

2) Speedup: It can be seen from Fig. 8 that the speedup remains almost the same when the dimension grows. The reason is that the parallelization is applied only to the population size in GPU-SPSO, but not to the dimension. Still, the function with more complex arithmetic has a higher speedup.

C. Other Characteristics of GPU-SPSO

1) Maximum Speedup: In some applications, large swarm population is needed during the optimization process. In this case, GPU-SPSO can greatly benefit the optimization by improving the running speed dramatically. Now we will carry through an experiment to find out the maximum speedup that GPU-SPSO can reach. We run GPU-SPSO and CPU-SPSO on f_3, respectively. Set D=50, $Iter$=10000, and both GPU-SPSO and CPU-SPSO are run only once (As the time needed for each run is almost the same). The result is shown in TABLE VIII. Global best solutions are obtained in both cases.

As shown in TABLE VIII that GPU-SPSO can reach a maximum speedup of greater than **11** when the swarm population size is 20000, running on f_3. And on the functions more complex than f_3, speedup may be even greater.

2) High Dimension Application: In some real world applications such as face recognition and fingerprint recognition, the problem dimension may be very high. Running PSO on CPU to optimize high dimensional problems is very slow, but the speed can be greatly accelerated if run it on GPU.

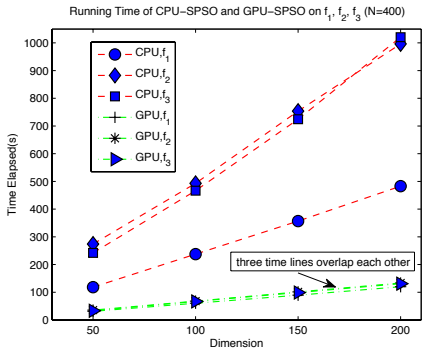

Fig. 7. Running Time and Dimension

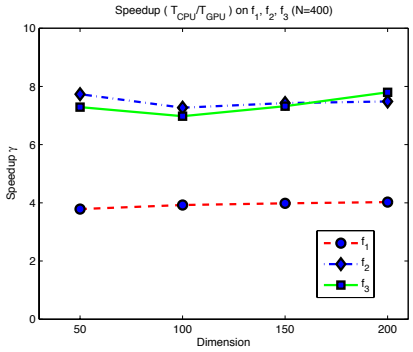

Fig. 8. Speedup and Dimension

Now we run both GPU-SPSO and CPU-SPSO on f_3 once, respectively (N=400, $Iter$=5000). The results are given in TABLE IX.

From TABLE IX, we can find that even when the dimension is as large as 3000, GPU-SPSO can run more than 6.5 times faster than GPU-SPSO.

3) Functions with Dependent Variables: As mentioned above, f_4 is one of the functions which have dependent variables. We run both GPU-SPSO and CPU-SPSO on f_4 for 10 times, and the results are shown in TABLE X. Seen from TABLE X, we can conclude GPU-SPSO can work quite well with functions with dependent variables, and reach a noticeable speedup.

D. Comparison with Related Works

The PSO algorithm was also implemented on GPU in an other way by making use of the texture-rendering of GPU [11]. We may call it **Texture-PSO**. This method used the textures on GPU chips to store particle information, and the fitness evaluation, velocity and position updating were

TABLE VIII

GPU-SPSO AND CPU-SPSO ON f_3 (D=50)

N	CPU-Time(s)	GPU-Time(s)	**Speedup**
10000	1269.7554	113.1295	**11.224**
20000	2537.7515	221.9755	**11.4326**

TABLE IX

GPU-SPSO AND CPU-SPSO ON f_3 (N=400)

D	CPU-Time(s)	GPU-Time(s)	**Speedup**
2500	1220.4361	92.1728	**6.5272**
3000	1461.7988	223.8946	**6.5290**

TABLE X
GPU-SPSO AND CPU-SPSO ON f_4 (D=50,$Iter$=10000)

N	CPU-Time	GPU-Time	**Speedup**	CPU-Value	GPU-Value
400	292.4788	65.7535	**4.4481**	5.3283	1.7313
1200	893.9783	168.5218	**5.3048**	1.3125	1.7196
2000	1.49E+03	280.8636	**5.3076**	0.1627	0.2029
2800	2.11E+03	370.2878	**5.6915**	0.2687	0.3046

done by means of texture rendering. But Texture-PSO has several disadvantages below which makes it almost useless when doing real-world optimization.

- The dimension must be set to a specific number, and it can not be changed unless redesigning the data structures.
- When the swarm population is small, for example smaller than 400, the running time of the Texture-PSO may be even longer than corresponding PSO that runs on CPU.
- The functions which can be optimized by Texture-PSO must have completely independent variables as a result of the architecture of GPU textures. So functions like f_4 can not be optimized by the Texture-PSO.

Instead of using the textures on GPU, we use the global memory to implement GPU-SPSO in this paper. Global memory is more like memory on CPU than textures do. So GPU-SPSO has overcome all the three disadvantages mentioned above:

- The dimension serves as a changeable parameter and it can be set to any reasonable numbers. High dimensional problems are also solvable with our GPU-SPSO.
- When the swarm population is small, for example smaller than 400, the speedup can still be bigger than 4 or 5.
- The functions with dependent variables such as f_4 can also be optimized by GPU-SPSO.

VI. CONCLUSIONS

In this paper, a novel way to implement SPSO on GPU is presented (GPU-SPSO), based on the software platform of CUDA from NVIDIA Corporation. GPU-SPSO has the following features:

- The running time of GPU-SPSO is greatly shortened over CPU-SPSO, while maintaining similar performance. And the speedup can be more than **11** on f_3 and other functions with more complex arithmetic. On GPU chips that have much more multi-processors than Geforce 8600GT used in this paper, for example Geforce 8800 series, the GPU-SPSO is expected to run tens of times faster than CPU-SPSO.
- The running time and swarm population size take a linear relationship. This is also true for running time and dimension. And it takes almost the same time for GPU-SPSO to optimize functions with different arithmetic complexity, while CPU-SPSO takes much more time to optimize functions with more complex arithmetic, with the same swarm population, dimension and number of iterations. Furthermore, function with more complex arithmetic has a higher speedup.
- The swarm population can be very large, and the larger the population is, the faster GPU-SPSO runs than CPU-SPSO. So GPU-SPSO can especially benefit optimizing with large swarm population.
- High dimensional problems and functions with dependent variables can also be optimized by GPU-SPSO, and noticeable speedup can be reached.
- Because most display card in current common PC has GPU chips, more researchers can make use of our parallel GPU-SPSO to solve their practical problems.

Because of these characteristics of GPU-SPSO, it can be applied to a large scope of practical optimization problems.

Our future research will focus on implementing genetic algorithm and other swarm intelligence algorithms in terms of similar methods presented in the paper. We will also try to put our GPU-SPSO onto real-world applications.

ACKNOWLEDGMENT

This work is supported by National Natural Science Foundation of China (NSFC), under grant number 60875080 and 60673020, and partially financially supported by the Research Fund for the Doctoral Program of Higher Education (RFDP) in China. This work is also in part supported by the National High Technology Research and Development Program of China (863 Program), with grant number 2007AA01Z453.

REFERENCES

[1] J. Kennedy, R. Eberhart, "Particle Swarm Optimization" IEEE International Conference on Neural Networks, Perth, WA, Australia, Nov. 1995, pp.1942-1948.
[2] I. Buck et. al, "Brook for GPUs: Stream Computing on Graphics Hardware" ACM, 2004, pp.777-786.
[3] NVIDIA, *NVIDIA CUDA Programming Guide1.1*, Chapter 1: Introduction to CUDA, 2007.
[4] D. Bratton, J. Kennedy, "Defining a Standard for Particle Swarm Optimization" IEEE Swarm Intelligence Symposium, April 2007, pp.120-127.
[5] R. Yang, G. Welch, "Fast Image Segmentation and Smoothing Using Commodity Graphics Hardware" Journal of Graphics Tools, special issue on Hardware-Accelerated Rendering Techniques, 2003, pp.91-100.
[6] K.E.H. et. al, "Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware" Proceeding of SIGGRAPH, 1999, pp.277-286.
[7] Z.W. Luo, H.Z. Liu and X.C. Wu, "Artificial Neural Network Computation on Graphic Process Unit" Proceedings of International Joint Conference on Neural Networks, Montreal, Canada, 2005.
[8] M. Macedonia, "The GPU Enters Computing's Mainstream" Entertainment Computing,October 2003, pp.106-108.
[9] NVIDIA, *NVIDIA CUDA Programming Guide1.1*, Chapter 2: Programming Model, 2007.
[10] W. B. Langdon, "A Fast High Quality Pseudo Random Number Generator for Graphics Processing Units" IEEE World Congress on Evolutionary Computation, 2008. pp.459-465.
[11] J.M. Li, et. al, "A parallel particle swarm optimization algorithm based on fine grained model with GPU accelerating" Journal of Harbin Institute of Technology, (In Chinese), Dec. 2006, pp.2162-2166.