International Journal of Artificial Intelligence,
ISSN 0974-0635; Int. J. Artif. Intell.
Autumn (October) 2011, Volume 7, Number A11
Copyright © 2011 by IJAI (CESER Publications)

GPU-Based Parallel Multi-objective Particle Swarm Optimization

You Zhou and Ying Tan[1]

[1]Key Laboratory of Machine Perception (Ministry of Education),Peking University.
Department of Machine Intelligence, School of Electronics Engineering and
Computer Science, Peking University, Beijing 100871, China
Phone: +86-10-62767611, Email: ytan@pku.edu.cn

ABSTRACT

In the recent years, multi-objective particle swarm optimization (MOPSO) has become quite popular in the field of multi-objective optimization. However, due to a large amount of fitness evaluations as well as the task of archive maintaining, the running time of MOPSO for optimizing some difficult problems may be quite long. This paper proposes a parallel MOPSO based on consumer-level Graphics Processing Units (GPU), which, to our knowledge, is the first approach of optimizing multi-objective problems via PSO on the platform of GPU. Experiments on 4 two-objective benchmark test functions are conducted. Compared with the CPU based sequential MOPSO, our GPU based parallel MOPSO is much more efficient in terms of running time, and the speedups range from 3.74 to 7.92 times. When the problem is large-scale, i.e. the dimension of the decision vector is large, the speedups can be bigger than 10 times. Furthermore, the experimental results show that the larger the size of the swarm is, the more nondominated solutions are found, the higher the quality of solutions are, and the bigger the speedup is.

Keywords: MOPSO, GPU, Parallel, Speedup.

Computing Classification System: I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods, and Search

1 Introduction

Multi-objective problems are very common in real-world optimization filed, of which the objectives to be optimized are normally in conflict with respect to each other. As a result, there is no single solution for them. Instead, several solutions with good trad-off among the objectives should be found out.

Particle swarm optimization (PSO) is a population based global optimization technique (Kennedy, Eberhart et al., 1995). PSO was originally designed to solve single objective optimization problems, but it can also be extended for multi-objective, resulting in many multi-objective particle swarm optimization approaches (Reyes-Sierra and Coello, 2006).

Parsopoulos and Vrahatis proposed a Vector Evaluated PSO (VEPSO) approach for multi-objective optimization (only two-objective problems) (Parsopoulos and Vrahatis, 2002). The whole swarm consisted of two subswarms. Each subswarm was evaluated according to one of

the two objectives but information coming from the other subswarm was used to determine the change of the velocities. Through this technique, the two subswarms purchased their own objective, meanwhile, the influence of the other objective was also imposed, thus a good tradeoff between the two objectives could be made. Compared with Vector Evaluated Genetic Algorithm (VEGA) (Schaffer, 1985), VEPSO for multi-objective optimization had an almost identical performance.
In this paper, we modify and parallelize the VEPSO approach for multi-objective optimization based on GPU, forming our GPU MOPSO. Experiments on 4 two-objective benchmark test functions are conducted, with the results thoroughly analyzed. Firstly, the performance of our GPU MOPSO is compared with the corresponding CPU based serial MOPSO in terms of running speed. Secondly, the benefits of implementing MOPSO on GPU are presented.
The rest of the paper is organized as follows: in Section 2, a short overview of multi-objective optimization as well as a brief introduction to GPU based computing are given, followed by the discussions of several approaches for parallelizing multi-objective optimizing methods; Section 3 describes the parallel implementation of GPU based MOPSO. Experimental results are reported and analyzed in Section 4, followed by conclusions in Section 5.

2 Related Works

2.1 Multi-objective Optimization

A general multi-objective optimization problem can be described as a vector function **f** that maps a tuple of D decision variables to a tuple of M objectives:

$$\begin{aligned} &min/max && \mathbf{y} = \mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), ..., f_M(\mathbf{x})) \\ &subject\ to && \mathbf{x} = (x_1, x_2, ..., x_D) \in \mathbf{X} \\ & && \mathbf{y} = (y_1, y_2, ..., y_M) \in \mathbf{Y} \end{aligned}$$

where **x** is called the decision vector, **X** is the parameter space, **y** is the objective vector, **Y** is the objective space. The set of solutions of a multi-objective optimization problem consists of all decision vectors for which the corresponding objective vectors cannot be improved in any dimension without degradation in another$-$these vectors are known as Pareto optimal.
Mathematically, the concept of Pareto optimality can be defined as follows: without loss of generality, assume a minimization problem and consider two decision vectors $\mathbf{a}, \mathbf{b} \in \mathbf{X}$. **a** is said to dominate **b** (written as $\mathbf{a} \prec \mathbf{b}$) iff

$$\forall i \in \{1, 2, ..., m\} : f_i(\mathbf{a}) \leqq f_i(\mathbf{b}) \wedge \exists j : f_j(\mathbf{a}) < f_j(\mathbf{b})$$

where $j \in \{1, 2, ..., m\}$ as well.
All decision vectors which are not dominated by any other decision vector of a given set are called nondominated regarding this set. The decision vectors that are nondominated within the entire search space are denoted as Pareto optimal and constitute the socalled Pareto-optimal set or Pareto-optimal front.

2.2 Multi-objective PSO

PSO is a population-based search algorithm based on the simulation of the social behavior of birds within a flock (Kennedy et al., 1995). It is a very popular global optimizer for problems in which the decision variables are real numbers.
First let us define the notations adopted in this paper. Assuming that the search space is D-dimensional, the position of the $i-th$ particle of the swarm is represented by vector $X_i = (x_{i1}, x_{i2}, ..., x_{iD})$ and the best particle in the swarm is denoted by the index g. The personal best position of the $i-th$ particle is represented as $\tilde{P}_i = (p_{i1}, p_{i2}, ..., p_{iD})$ and the velocity of the i-th particle is recorded as $V_i = (v_{i1}, v_{i2}, ..., v_{iD})$, and t is the number of current iteration. Following these notations the particles are manipulated according to the following equations iteratively:

$$\begin{aligned} v_{id}(t+1) =& w \cdot v_{id}(t) + c_1 \cdot r_1(p_{id}(t) - x_{id}(t)) \\ & + c_2 \cdot r_2(p_{gd}(t) - x_{id}(t)) \end{aligned} \tag{2.1}$$

$$x_{id}(t+1) = x_{id}(t) + v_{id}(t+1) \tag{2.2}$$

where $d = 1, 2, ..., D$; $i = 1, 2, ..., N$, here N is the size of the population. w is the inertia weight; c_1 and c_2 are two positive constants; r_1 and r_2 are two random numbers within the range [0,1]. In this paper, w is initially set to 1.0 and linearly decreases to 0.4, $c_1 = c_2 = 2.05$. If x_{id} exceeds the boundary limitation X_{max} or X_{min}, it will be directly set to X_{max} or X_{min}.
In order to solve multi-objective optimization problems, the original PSO has to be adjusted, as the solution set of a problem consists of different solutions (a Pareto-optimal front) instead of a single one. Given the population-based nature of PSO, it is desirable to produce several (different) nondominated solutions within a single run. The leaders should be carefully chosen among the nondominated solutions found so far, which are usually stored in an external archive. There are currently over twenty five different proposals of MOPSOs reported. A survey of this specialized literature was given by Coello in (Reyes-Sierra and Coello, 2006).
In 2002, Parsopoulos proposed a VEPSO approach for multi-objective optimization (Parsopoulos and Vrahatis, 2002), which might be the first particle swarm approach for multi-objective optimization. They used two subswarms to solve two-objective problems. Each subswarm was evaluated with only one of the two objectives. Meanwhile, the best particle coming from the other subswarm was used for the determination of the new velocities of its own particles. The experimental results showed that the performance of this approach was as good as VEGA, which was a well-known evolutionary algorithm approach for multi-objective optimization.
They extended their VEPSO approach to multi-objective problems with more than two objectives in 2004 (Parsopoulos, Tasoulis, Vrahatis et al., 2004). They used M subswarms to optimize S objectives (M may be bigger than S). Each subswarm optimizes a single objective, and they exchange their best particles through a ring topology. But the performance of this model was only tested on two-objective problems.
Compared with other MOPSO methods, the VEPSO approach had the following promising features.

2.2.1 Easy Implementation

As the non-dominated solutions found during the evolution process are not responsible for the guidance of the particles during the evolution process, they are stored in the archive just for outcome. Strategies which focus on choosing good solutions out of the archive to guide the whole swarm are not necessary, so the implementation of VEPSO is relatively simple.

2.2.2 Suitable for Parallelization

It can be parallelized based on various parallel hardware architecture, e.g. multi-computer or multi-processor systems, thus the running speed of VEPSO can be greatly accelerated.

2.3 Graphics Processing Units and CUDA

Graphics Processing Units (GPU) was originally designed specifically for image and graphic processing, where compute-intensive and highly parallel computing is required. GPU computes faster than CPU and 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. Because of these advantages, general-purpose computing through GPU is becoming a new trend.
Recently, NVIDIA developed the Compute Unified Device Architecture (CUDA^{TM}) technology which enables researchers to implement their GPU-based applications more easily (CUDA^{TM}, 2008). It is a new hardware and software architecture which runs under C language environment for the purpose of managing computations on the GPU as a data-parallel computing device, without the need of mapping them to a graphics API.
Two important concepts of CUDA^{TM}are *thread batching* and *memory model*.

2.3.1 Thread Batching

When programming through CUDA, the GPU is viewed as a compute device capable of executing a very large 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 ID of the thread with the index (x,y) is $y * D_x + x$.

2.3.2 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 and read-only per-grid texture memory.

2.4 Parallel Approaches for Multi-objective Optimization

Until now, most of the parallel approaches for multi-objective optimization are evolutionary algorithms. The implementation of those algorithms are usually based on MIMD (Multiple Instruction, Multiple Data) architectures, which are either multicomputer systems such as grid computing and clusters of workstations, or multiprocessor systems such as symmetric multiprocessors (SMP) and non-uniform memory access (NUMA) system. There are several parallelization models (Jaimes and Coello, 2009).

2.4.1 Master-Slave Model

One of the simplest ways to parallelize a multi-objective evolutionary algorithms (MOEA). A master processor executes the MOEA, and the objective function evaluations are distributed among a number of slave processors. Once the slaves have completed the evaluations, the objective function values are returned to the master. Using this model of parallelization, the running time can be greatly reduced depending on the number of slaves, meanwhile, the same solutions can be found by its serial counterpart.

2.4.2 Diffusion Model

A unique population is considered, and it is spatially distributed onto a neighborhood structure, which is usually a two-dimensional rectangular grid, with one individual on each grid point. This model is appropriate for shared-memory MIMD computers such as SMPs. However, custom hardware implementations on SIMD computers are also possible.

2.4.3 Island Model

The population is divided into several small sub-populations, called islands or demes, which are independent of each other. In each island a serial MOEA is executed for a number of generations, and after that, individuals migrate between neighboring islands, according to some migration topology. This model is well-suited for clusters of computers or for grid computing systems.

2.4.4 Hybrid Models

This model combines a coarse-grained parallel scheme at a high level (e.g., island model) with a fine-grained scheme at a low level (e.g., diffusion model).
There are also some approaches for parallelizing MOEA utilizing SIMD architectures such as Graphics Processing Unit (GPU). Wong (Wong, 2009) implemented a parallel MOEA based on GPU within the environment of CUDA$^{\mathrm{TM}}$. The speedups of the parallel MOEA range from 5.62 to 10.75 times.
Parallel MOPSO are not so well studied as parallel MOEA, and only a very limited number of parallel MOPSO approaches can be found in the literature. Mostaghim et al. (Mostaghim, Branke and Schmeck, 2007) designed two methods to parallelize MOPSO: cluster-based MOPSO

and hypervolume-based MOPSO, both of which worked on several number of processors, namely computer grids. The basic idea of these methods was to divide the population into subswarms which can be processed in parallel. The parallel VEPSO proposed by Parsopoulos (Parsopoulos et al., 2004) mentioned in 2.2 was another example of parallel MOPSO, which was implemented on the parallel virtual machine (PVM).
In our previous work (Zhou and Tan, 2009), we implemented standard PSO in parallel based on GPU using CUDA^{TM}, and a maximum speedup of 10 times was achieved on some functions. In this paper, we intend to design a GPU-based MOPSO method on the basis of VEPSO, which, to our knowledge, is the first attempt of conducting MOPSO on GPU.

3 GPU-based MOPSO

In VEPSO, the whole swarm consists of two subswarms with the same size. Assume that the total swarm size is N, then the population of each subswarm is $N/2$. For each subswarm, Equation(2.1) (velocity update process) should be adjusted as follows:

$$\begin{aligned} v_{id}(t+1) =& w \cdot v_{id}(t) + c_1 \cdot r_1(p_{id}(t) - x_{id}(t)) \\ &+ c_2 \cdot r_2(p^*_{gd}(t) - x_{id}(t)) \end{aligned} \tag{3.1}$$

where p^*_{gd} is the $d-th$ element of the global best particle's position vector coming from the other subswarm. In this way, each subswarm updates the velocities and positions of its particles using Equation(3.1) and (2.2) corresponding to its own objective, while the influence of the other objective is also imposed. The nondominated solutions found in each iteration are stored in an external archive, which is built and maintained using the pseudo-code provided in (Jin, Olhofer and Sendhoff, 2001).
In order to parallelize VEPSO according to the specific hardware architecture of GPU, we modify VEPSO and propose our parallel GPU based MOPSO (ab. GPU MOPSO). In GPU MOPSO, the on-line performance is considered instead of off-line performance utilized in VEPSO, which means there is no need to maintain an external archive. On-line performance means that only the nondominated solutions in the final population are considered as the outcome, while off-line performance takes the nondominated solutions generated during the entire evolution process into account (Zitzler, Deb and Thiele, 2000). The following benefits can be obtained by this modification:

- Each particle in the swarm is evaluated by only one of the two objectives, which halves the computational task of fitness evaluation, while in original VEPSO, each particle must be evaluated using both of the objectives.
- The time-consuming data transferring between GPU and CPU can be greatly reduced, as there is no need to transfer the GPU-resided fitness and position values of each nondominated solution back to CPU, resulting in a higher speedup.

The proposed GPU MOPSO is described by Algorithm 1. In this algorithm, $Iter$ denotes the maximum number of iterations, and $i = 1, 2, ..., N$. The fitness values of all the N particles are

Algorithm 1 Algorithm for GPU MOPSO

```
1: Set t, the generation counter to 0.
2: Generate $N$ individuals for initial population.
3: Transfer initial data from CPU to GPU.
4: for $t$=0 to $Iter$ do
5:     Evaluate_Fitness()
6:     Update_Pbest()
7:     Transfer $F$ back to CPU, find out the indices ($g$) of the best particle in each subswarm
       via $F$.
8:     Update velocity and position values of the $i-th$ particle, using Equation( 3.1) and
       ( 2.2).
9: end for
10: Transfer data back to CPU, evaluate each particle using both of the objectives, output the
    nondominated solutions.
```

stored in a 1-dimensional array F. Steps 5, 6, 8 are executed in parallel, employing N threads, each of which is allocated with one particle to process.
In the following subsections, the details for implementing GPU MOPSO are given.

3.1 Data Organization

As there are N individuals and each one is represented by a D-dimensional vector, we use a 1-dimensional array X of size $N * D$ on global memory to store the whole population.
When the concurrent memory accesses by CUDA threads in a half wrap (16 parallel threads) can be coalesced into a single memory transaction, the global memory bandwidth can be improved. In order to fulfill the requirements for coalesced memory accesses, the same variables from all individuals are grouped and form a tile of N values in the global memory. The representation scheme for the whole swarm is shown in Fig. 1.

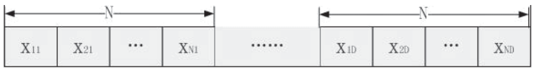

Figure 1: Representation of N individuals with D variables on global memory

Generating random numbers on GPU are very tricky though there are several existing approaches (Howes and Thomas, 2007; Pang, Wong and Heng, 2008). In order to focus on the implementation of MOPSO, we would rather generate random numbers on CPU and transfer them to GPU. For the purpose of saving transferring time, we do it in the following way: Q $(Q >> D * N)$ random numbers are generated on CPU before running GPU MOPSO. Then they are transferred to GPU once for ado and stored in an array on the global memory. Each time random numbers are needed during the evolution process, pass a random integer (serves as a start point) to GPU, and fetch the corresponding number of random numbers from global memory for use. This may impose some negative influence on the performance of GPU MOPSO, as the random numbers are not exactly "random".

3.2 Fitness Value Evaluation

The computation of fitness values is the most important task during 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 of our GPU MOPSO. It is not necessary to get the complete objective vector for each individual in GPU MOPSO. Assume that the two objectives of the optimization problem are f_1 and f_2. f_1 is used to evaluate the individuals of the first subswarm (the first $N/2$ particles), and f_2 for the remaining $N/2$ particles. As there are no interaction among threads, the fitness value evaluation process can be fully parallelized. The fitness values of all the particles are stored in F, which is an array of size N.
In step 5 of Algorithm 1, the fitness value of each individual is computed in the way as shown in Algorithm 2. We can see that the iterations are only applied to dimension index $d = 1, 2..., D$. The reason is that the arithmetical operations on all the N data of all the particles in dimension d is done in parallel (synchronously) on GPU.

Algorithm 2 Evaluate_Fitness()

```
1: Initialize, set the 'block size' and 'grid size', with the number of threads in a grid equaling
   to the number of particles ($N$).
2: for each dimension $d$ do
3:     Map all the $N$ threads to the $N$ position values of dimension $d$ one-to-one
4:     Load $N$ data from global to shared memory
5:     Apply arithmetical operations of $f_1$ to the first $N/2$ data and $f_2$ to the remaining $N/2$
       data in parallel
6:     Store the result of dimension $d$ with $F_d$
7: end for
8: Combine $F_d$ ($d = 1, 2...D$) to get the final fitness values of all particles, store them in array
   $F$.
```

There are two instructions for mapping all the threads to the N data in a one-dimensional array (step 3 of Algorithm 2), which are listed as bellow:

- 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 I-th datum in a one-dimensional array, then the relationship between the indices and I is:

$$I = (B_y * T_2 + B_x) * S_1 * S_2 + T_y * S_2 + T_x \tag{3.2}$$

In this way, all the threads in a kernel are mapped to N data one to one. Operations are applied to all the N threads synchronously. This is the main mechanism for accelerating the computing.

3.3 Update of Personal and Global Best Particle

After the fitness values are updated, each particle may arrive at a better position $\tilde{P}$ than ever before and a new global best position may be found. So $\tilde{P}$ and the index g (refer to equation. 3.1) must be updated according to the current status of the particle swarm. The updating process of $\tilde{P}$ can be achieved by Algorithm 3, in which PX and PF store the positions and fitness values of personal best particles, respectively. $i = 1, 2, ..., N$ is the index of thread, which are executed in parallel, and $X(d * N + i)$ is the position value of particle i on dimension d.

Algorithm 3 Update_Pbest()

```
1: Map all the threads to $N$ particles one-to-one.
2: Transfer all the $N$ data from global to shared memory.
3: if $F(i)$ is better than $PF(i)$ then
4:     $PF(i)$= $F(i)$
5:     for each dimension $d$ do
6:         Store position $X(d * N + i)$ with $PX(d * N + i)$
7:     end for
8: end if
```

The global best particle update procedure (step 7 in Algorithm 1) is performed on CPU, as the minimum (or maximum) fitness values in each subswarm must be found in array F. Although it is possible to execute it on GPU, it is quite complex to implement and not efficient enough when N is relatively small. We transfer F back to CPU, and the indices (g) of the best particle in each sub-swarm are found and recorded, which will be passed to the velocity and position update procedure as parameters.

3.4 Update of Velocity and Position

The update of velocity and position (step 8 in Algorithm 1) for the whole swarm is an essential procedure in GPU MOPSO and it can be fully parallelized. The position of the global best particle ($\tilde{P_g}$) of the second subswarm is responsible for the velocity update of the first subswarm, and vice versa.

3.5 Selection of Nondominated Solutions

In the final generation, the nondominated solutions are picked out from the entire population, and they are returned as outcome. As there are N individuals, multiple solutions may exist, from which pareto fronts are constructed.

4 Experiment and Results Analysis

The benchmark test functions of this paper are adopted from (Parsopoulos and Vrahatis, 2002), which are listed in TABLE 1. Deb (Deb, Thiele, Laumanns and Zitzler, 2005) gives a comprehensive study on how to construct test functions in 2005.

Table 1: Two-objective Test Functions

Function	m	Equation	bounds	D
F1	2	$f_1 = \frac{1}{D}\sum_{i=1}^{D} x_i; f_2 = \frac{1}{D}\sum_{i=1}^{D}(x_i - 2)^2$	$[0,1]$	30
F2	2	$f_1 = x_1; f_2 = g * h; g = 1 + 9.0\sum_{i=2}^{D} x_i/(D-1);$ $h = 1 - \sqrt{f_1/g}$	$[0,1]$	30
F3	2	as F2, except $h = 1 - \sqrt{f_1/g} - (f_1/g)sin(10\pi * f_1)$	$[0,1]$	30
F4	2	as F2, except $h = 1 - \sqrt[4]{f_1/g} - (f_1/g)^4$	$[0,1]$	30

We have also implemented a serial CPU based MOPSO (ab. CPU MOPSO) approach corresponding to GPU MOPSO, just for comparison. All of the experiments are conducted on CUDA^{TM} platform, based on an Intel Core 2 Duo 2.20 GHz CPU, 3.0G RAM machine. The display card is NVIDIA Geforce 9800GT, and OS is Windows XP.
We set $N = 1024, 2048, 4096, 8192$, respectively. Both GPU MOPSO and CPU MOPSO are executed for 30 times to optimize the four test functions with $Iter = 250$. For each function, the average number of nondominated solutions (denoted as No.S) found by these two approaches, as well as the average running time are recorded. All the results are listed in TABLE 2, 3, 4, 5. Figs. 2, 3, 6 are drawn to depict the data in the tables for the purpose of a more vivid impression.

Table 2: Results of CPU MOPSO and GPU MOPSO on F1

N	CPU MOPSO		GPU MOPSO		**Speedup**
	No.S	Time	No.S	Time	
1024	40	1.01	39	0.21	**4.76**
2048	49	2.03	45	0.33	**6.18**
4096	51	4.38	37	0.62	**7.06**
8192	79	9.40	42	1.19	**7.92**

Table 3: Results of CPU MOPSO and GPU MOPSO on F2

N	CPU MOPSO		GPU MOPSO		**Speedup**
	No.S	Time	No.S	Time	
1024	22	0.97	24	0.26	**3.74**
2048	41	1.98	30	0.45	**4.36**
4096	95	4.48	147	0.85	**5.23**
8192	171	9.56	106	1.69	**5.65**

Table 4: Results of CPU MOPSO and GPU MOPSO on F3

N	CPU MOPSO		GPU MOPSO		**Speedup**
	No.S	Time	No.S	Time	
1024	15	0.95	13	0.25	**3.84**
2048	19	1.90	15	0.46	**4.14**
4096	23	3.96	19	0.85	**4.69**
8192	29	9.30	24	1.66	**5.59**

Table 5: Results of CPU MOPSO and GPU MOPSO on F4

N	CPU MOPSO		GPU MOPSO		**Speedup**
	No.S	Time	No.S	Time	
1024	33	0.99	30	0.26	**3.78**
2048	52	2.02	78	0.47	**4.29**
4096	125	4.60	121	0.86	**5.36**
8192	208	9.71	291	1.80	**5.40**

4.1 Running Time and Speedup VS. Swarm Size

Fig. 2 shows the relationship between running time and the swarm size (N), when running CPU MOPSO and GPU MOPSO to optimize function $F_1 - F_4$, respectively. As N grows in an exponential way, the running time of CPU MOPSO also grows in the exponential way, while the running time of GPU MOPSO increases in a linear way.
Fig. 3 depicts how speedup changes with N. As N grows, the speedup of the GPU MOPSO over CPU MOPSO also increases, running on $F_1 - F_4$. The speedups range from $3.74 - 7.92$, and when N is bigger than 8192, the speedups can be expected to be even bigger.

4.2 Pareto Fronts Found by GPU MOPSO and CPU MOPSO

The Pareto-optimal fronts of $F_1 - F_4$ found by GPU MOPSO and CPU MOPSO are shown in Fig. 4 and Fig. 5, respectively. Each of them are constructed by the specific run selected from the 30 runs which returns the most number of nondominated solutions in the final generation. It can be seen from the figures that the pareto fronts found by GPU MOPSO is as good as those found by CPU MOPSO, both with high diversities. In fact, these fronts are as good as the fronts returned by the original VEPSO (Parsopoulos and Vrahatis, 2002).

4.3 Quantity and Quality of Solutions VS. Swarm Size

In single objective optimization, a larger swarm size does not certainly mean a better optimizing results, which was already pointed out by Bratton in (Bratton and Kennedy, 2007), and a swarm size of $20 - 100$ was suggested. Consequently, it is not quite necessary to implement PSO on

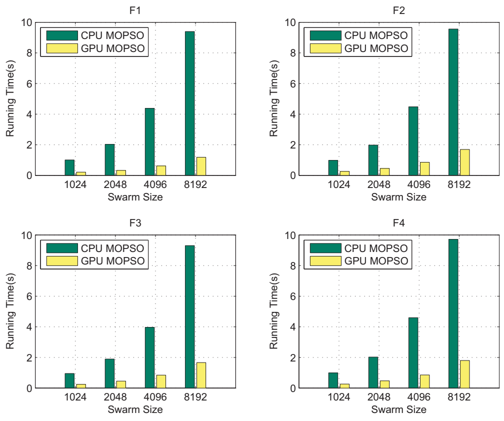

Figure 2: Running Time VS. Swarm Size

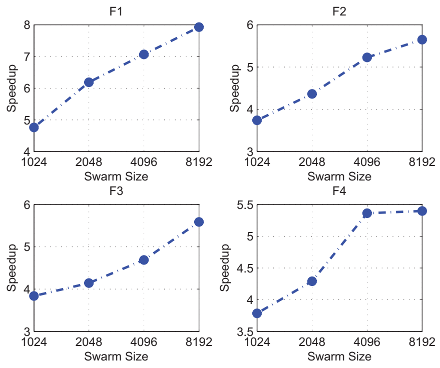

Figure 3: Speedup VS. Swarm Size

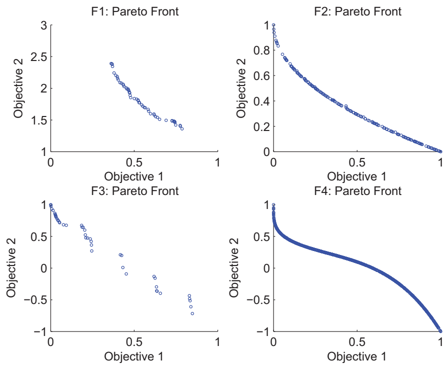

Figure 4: Pareto fronts found by GPU MOPSO (N=4096)

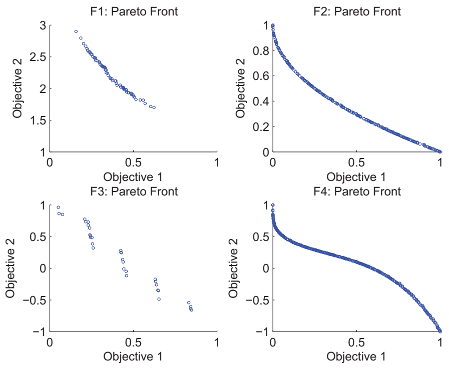

Figure 5: Pareto fronts found by CPU MOPSO (N=4096)

GPU in single objective optimization, as the GPU based parallel PSO with a small swarm (with $20-100$ individuals) may even run slower than serial PSO on CPU.

However, in MOPSO, a larger swarm may be more powerful in searching for nondominated solutions. As can be seen from Fig. 6, as the swarm size (N) grows, the number of nondominated solutions found by CPU MOPSO also increases, when running on all of the four test functions. While in GPU MOPSO, this is also true when running on F_2-F_4, but not so convictive on F_1. The reason may be that the random numbers used by GPU MOPSO are not quite exactly "random", for which the reason is explained in Section 3.1.

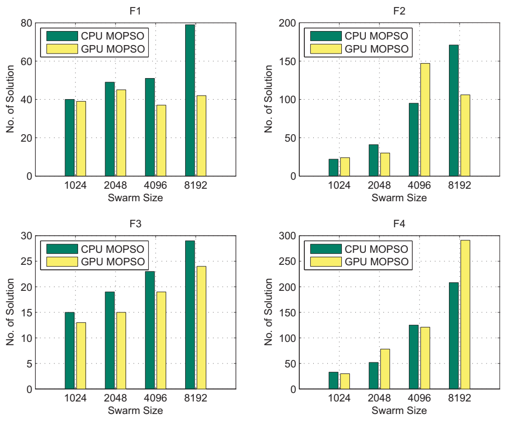

Figure 6: Number of Solutions VS. Swarm Size

Large swarm size is not only beneficial to finding larger quantity of nondominated solutions, but also helps in improving the quality of them.

The obtained solutions of all 30 runs are evaluated using an established measure, the $\mathcal{C}$ metric (Zitzler et al., 2000). The metric $\mathcal{C}(A,B)$ measures the fraction of members of the Pareto-optimal front B that are dominated by members of the Pareto-optimal front A. Notice that the $\mathcal{C}$ metric is neither symmetrical in their arguments nor satisfies the triangle inequality, thus $\mathcal{C}(A, B) \neq \mathcal{C}(B, A)$.

Here we use $\mathcal{C}(N_1, N_2)$ to denote the $\mathcal{C}$ values of pareto fronts returned by the GPU MOPSO swarm of size N_1 and N_2. The $\mathcal{C}$ values are statistically displayed with matlab boxplots in Fig. 7. Each boxplot represents the distribution of the $\mathcal{C}$ values for the ordered pair $(2048, 1024)$ and $(1024, 2048)$, respectively.

Each box of the boxplot has lines at the lower quartile, median, and upper quartile values. The lines that extend from each end of the box are the whiskers, which show the extent of the rest of the data. The outliers that lie beyond the ends of the whiskers are displayed with a red + sign.

As can be seen from Fig. 7, the nondominated solutions found by the swarm of size 2048 dominate a bigger fraction of the solutions obtained by the swarm of size 1024. Contrarily, the fraction is much lower. It can be concluded that the solutions found by the swarm of bigger size are more quality than those obtained by a small population.

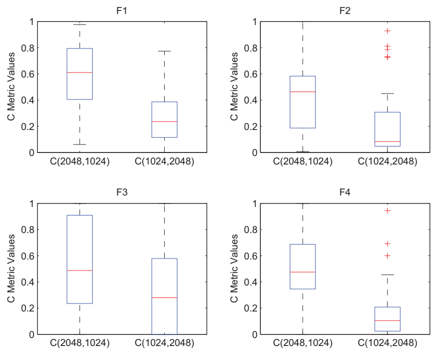

Figure 7: Quality Metric of Nondominated Solutions

4.4 GPU MOPSO for Large-scale Multi-objective Problems

We call a multi-objective problem as "large-scale", when the dimensions of either the decision vector or the objective vector are relatively large. Here we run the CPU MOPSO and GPU MOPSO, respectively, for optimizing $F4$ with large dimension. The size of the swarm is set to 4096, and the dimension of the decision vector $D = 100, 200$, respectively. The other settings are the same with previous experiments. The experimental results are list in TABLE 6. From this table, we can see that when the dimension is relatively large, the GPU MOPSO can even run more than 10 times faster than the corresponding CPU MOPSO, while maintaining comparable optimizing performances.

Table 6: Results of CPU MOPSO and GPU MOPSO on F4 ($N = 4096$)

D	CPU MOPSO		GPU MOPSO		**Speedup**
	No.S	Time	No.S	Time	
100	26	29.23	34	2.80	**10.44**
200	34	60.63	46	5.67	**10.69**

4.5 Comparison with Related Works

Wong (Wong, 2009) implemented a parallel MOEA based on GPU within the environment of CUDA^{TM}. This algorithm contained five steps: fitness computation, parent selection, crossover and mutation, dominance checking and non-dominated selection. All five procedures were performed on GPU except the non-dominated selection procedure, and the implementation was quite complex. The speedups of the parallel MOEA ranged from 5.62 to 10.75 times, but the optimizing performances (e.g. number of non-dominated solutions, pareto fronts) were not given.
Compared with the parallel MOEA, our parallel GPU MOPSO is easier for implementation, while keeping a promising optimizing performance as well as comparable speedups.

5 Conclusions and Future Works

In this research, for the first time in the MOPSO research field, we implemented a parallel MOPSO based on GPU (specifically, only two-objective problems are considered), within the platform of CUDA, resulting in our GPU MOPSO method. It has the following promising features:

- Each particle in the swarm is evaluated by only one of the objectives instead of both, thus the time consuming fitness computation task is halved, and the running time is greatly shortened.
- The Pareto-optimal fronts are constructed by the nondominated solutions selected from the last generation of the swarm, thus the maintaining of an external archive for non-dominated solutions is not necessary, which is a complex and quite time consuming procedure. The Pareto-optimal fronts of the four test functions found by GPU MOPSO are quite close to the true fronts, with high diversities.
- The bigger the size of the swarm is, the more nondominated solutions are found, the higher their quality are (closer to the true fronts of the problems), and the bigger speedup is reached by GPU MOPSO. The speedups range from 3.74 to 7.92, depending on the functions to be optimized and the size of the swarm.
- GPU MOPSO performs well on the functions with large dimensions, i.e. large-scale problems, and the speedups are bigger than 10 times.

- This is the first approach to implement MOPSO methods based on consumer-level GPU, which is available on common PC. Compared with other implementations which are based on multicomputer or multiprocessor systems, such as clusters, the hardware requirement is much lower.

For future work, we will extend our GPU MOPSO to multi-objective optimization problems with more than two objectives, and the possibility of designing new migration schemes for communicating information among the subswarms will be studied.

6 Acknowledgement

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

Bratton, D. and Kennedy, J. 2007. Defining a standard for particle swarm optimization, *Swarm Intelligence Symposium, 2007. SIS 2007. IEEE*, Honolulu, Hawaii, pp. 120–127.

CUDA^{TM}, N. 2008. *CUDA Programming Guide Version 2.2*.
*http://developer.nvidia.com/object/cuda.html

Deb, K., Thiele, L., Laumanns, M. and Zitzler, E. 2005. Scalable test problems for evolutionary multiobjective optimization, *in Evolutionary Multiobjective Optimization Theoretical Advances and Applications (A.Abraham, L. Jain and R. Goldberg, Eds.), Springer-Verlag, Berlin, Heidelberg, New York*, pp. 105–145.

Howes, L. and Thomas, D. 2007. Efficient random number generation and application using CUDA, *GPU gems,Addison Wesley* **3**: 805–830.

Jaimes, A. and Coello, C. 2009. Applications of Parallel Platforms and Models in Evolutionary Multi-Objective Optimization, *Biologically-Inspired Optimisation MethodsAndrew Lewis, Sanaz Mostaghim, and Marcus Randall (Eds.),Springer-Verlag, Berlin, Heidelberg* pp. 23–49.

Jin, Y., Olhofer, M. and Sendhoff, B. 2001. Dynamic weighted aggregation for evolutionary multi-objective optimization: Why does it work and how?, *Proceedings of The Genetic and Evolutionary Computation Conference(GECCO-2001)*, pp. 1042–1049.

Kennedy, J., Eberhart, R. et al. 1995. Particle swarm optimization, *Proceedings of IEEE international conference on neural networks*, Vol. 4, Perth, Australia, pp. 1942–1948.

Mostaghim, S., Branke, J. and Schmeck, H. 2007. Multi-objective particle swarm optimization on computer grids, *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, London, England UK, pp. 869–875.

Pang, W., Wong, T. and Heng, P. 2008. Generating massive high-quality random numbers using GPU, *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, Hong Kong, China, pp. 841–847.

Parsopoulos, K., Tasoulis, D., Vrahatis, M. et al. 2004. Multiobjective optimization using parallel vector evaluated particle swarm optimization, *Proceedings of the IASTED international conference on artificial intelligence and applications (AIA 2004)*, Vol. 2, Innsbruck, Austria, pp. 823–828.

Parsopoulos, K. and Vrahatis, M. 2002. Particle swarm optimization method in multiobjective problems, *Proceedings of the 2002 ACM symposium on applied computing*, Madrid, Spain, pp. 603–607.

Reyes-Sierra, M. and Coello, C. 2006. Multi-objective particle swarm optimizers: A survey of the state-of-the-art, *International Journal of Computational Intelligence Research,Citeseer* **2**(3): 287–308.

Schaffer, J. 1985. Multiple objective optimization with vector evaluated genetic algorithms, *Proceedings of the 1st International Conference on Genetic Algorithms*, Hillsdale, NJ, USA, pp. 93–100.

Wong, M. 2009. Parallel multi-objective evolutionary algorithms on graphics processing units, *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, Montreal, Qubec, Canada, pp. 2515–2522.

Zhou, Y. and Tan, Y. 2009. GPU-based parallel particle swarm optimization, *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, Trondheim, Norway, pp. 1493–1500.

Zitzler, E., Deb, K. and Thiele, L. 2000. Comparison of multiobjective evolutionary algorithms: Empirical results, *Evolutionary computation,MIT Press Cambridge, MA, USA* **8**(2): 173–195.