Optimization Methods and Software

Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/goms20

Accelerating parallel particle swarm optimization via GPU

Yukai Hung & Weichung Wang

Department of Mathematics, National Taiwan University, Taipei 106, Taiwan, Republic of China

Published online: 25 Aug 2010.

To cite this article: Yukai Hung & Weichung Wang (2012): Accelerating parallel particle swarm optimization via GPU, Optimization Methods and Software, 27:1, 33-51

To link to this article: http://dx.doi.org/10.1080/10556788.2010.509435

PLEASE SCROLL DOWN FOR ARTICLE

Full terms and conditions of use: http://www.tandfonline.com/page/terms-and-conditions

This article may be used for research, teaching, and private study purposes. Any substantial or systematic reproduction, redistribution, reselling, loan, sub-licensing, systematic supply, or distribution in any form to anyone is expressly forbidden.

The publisher does not give any warranty express or implied or make any representation that the contents will be complete or accurate or up to date. The accuracy of any instructions, formulae, and drug doses should be independently verified with primary sources. The publisher shall not be liable for any loss, actions, claims, proceedings, demand, or costs or damages whatsoever or howsoever caused arising directly or indirectly in connection with or arising out of the use of this material.
Accelerating parallel particle swarm optimization via GPU

Yukai Hung and Weichung Wang*

Department of Mathematics, National Taiwan University, Taipei 106, Taiwan, Republic of China

(Received 18 November 2009; final version received 14 July 2010)

Particle swarm optimization (PSO) is a population-based stochastic and derivative-free method that has been used to solve various optimization problems due to its simplicity and efficiency. While solving high-dimensional or complicated problems, PSO requires a large number of particles to explore the problem domains and consequently introduces high computational costs. In this paper, we focus on the acceleration of PSO for solving box-constrained, load-balanced optimization problems by parallelization on a graphics processing unit (GPU). We propose a GPU-accelerated PSO (GPSO) algorithm by using a thread pool model and implement GPSO on a GPU. Numerical results show that the GPU architecture fits the PSO framework well by reducing computational timing, achieving high parallel efficiency and finding better optimal solutions by using a large number of particles. For example, while solving the 100-dimensional test problems with $65,536 (16 \times 2^{12})$ particles, GPSO has achieved up to $280X$ and $83X$ speedups on a NVIDIA Tesla C1060 1.30 GHz GPU relative to an Intel Xeon-X5450 3.00 GHz central processing unit running in single- and quad-core mode, respectively. GPSO provides a promising method for tackling high-dimensional and difficult optimization problems using a low-cost and many-core GPU system.

Keywords: particle swarm optimization; parallel computing; graphic process unit

1. Introduction

Particle swarm optimization (PSO) is a derivative-free and population-based stochastic optimization method. One focus of PSO research is the development of efficient algorithms that can reduce the computational time for finding desired optimal function values. In this article, we use a general purpose graphic processing unit (GPU) as the computational tool to pursue this goal. In particular, we develop and implement parallel algorithms on a system equipped with a central processing unit (CPU) and an NVIDIA Tesla C1060 GPU to efficiently solve the following (high-dimensional) minimization problem with simple box constraints.

$$\min_{x \in \mathbb{R}^D} f(x),$$

s.t. $l_j \leq x_j \leq u_j$, for $j = 1, \ldots, D$, \hspace{1cm} (1)

where $f: \mathbb{R}^D \to \mathbb{R}$ is the objective function, $D$ is the dimension of the search space, and $l_j$ and $u_j$ are the lower and upper bounds of the $j$th component of $x$, respectively.

*Corresponding author. Email: wwang@math.ntu.edu.tw

ISSN 1055-6788 print/ISSN 1029-4937 online
© 2012 Taylor & Francis
http://dx.doi.org/10.1080/10556788.2010.509435
http://www.tandfonline.com
The original idea of PSO arises in a simplified social swarm model. Each particle of the swarm, in analogy to a fish or a bird, searches for an optimum based on a sequence of stochastic velocities that are determined by its own experience and the knowledge of the whole swarm. Ever since its debut in 1995 [24], a great amount of theoretical research has been conducted to solve unconstrained, constrained, multi-solution, multi-objective, dynamic, and discrete optimization problems. Additionally, PSO has been successfully applied to a broad range of application areas, including structure design, economic dispatch, supply chains, geotechnical engineering, and many others. Survey articles [1,2,35] and books [3,8,13,36] discuss PSO in detail and provide further references.

Among the attempts that are made to accelerate overall execution time and increase solution robustness, parallelism in computations [15,26,37,41,43] is one of the primary focuses. The main motivations for parallelism are the fact that each particle can explore for optimality independently and the demand for a large number of particles in high-dimensional problems. Issues such as scalability, synchronous or asynchronous implementation, coherence, and network communication should be addressed for parallel PSO [37]. Other efforts for improving parallel efficiency include communication strategies for problems with weak or strong correlated parameters [5], usage of the MapReduce programming model [10] for moderately difficult problems [29], and a multi-server scheme for a large number of processors [41]. For parallel computer architecture, most of the proposed parallel PSO algorithms are based on computer clusters with distributed memory using a message passing interface for communication. Few parallel PSO implementations are based on OpenMP [42]. Furthermore, it is worth noting that an analysis in [8] indicates that a version of parallel PSO (updating particle information in batches) does not require more function evaluations than sequential PSO (updating particle information one-by-one) but has a shorter total duration time.

While PSO is natural for parallelism at the algorithmic level, it is not straightforward for achieving full acceleration in the corresponding implementation. The main issues here are data communication and load balancing, and these two issues are correlated. In PSO, most of the computational cost is devoted to function evaluations. If function evaluations are costly (in a relative sense), the data communication cost may be ignored; thus, the main issue to be resolved is load balancing. In contrast, if function evaluations are relatively cheap, data communication efficiency dominates the parallel scalability.

Note that fault tolerance is another issue for large-scale homo- or heterogeneous clusters. However, this issue is beyond the scope of this paper.

Both synchronous and asynchronous parallel PSOs have been proposed. For synchronous parallel PSO, all processors wait until all of the function evaluations are done before moving to the next steps. Several numerical experiments considering synchronous parallel PSO report results in the form of (parallel efficiency, processor numbers) that range from (80%, 5) to (40%, 20) [37], (85%, 5) to (40%, 20) [26], (30%, 17) to (17%, 47) [43], and (63%, 4) to (40%, 32) [41]. Note that the parallel efficiency refers to the ratio of acceleration to the number of processors. The results suggest that parallel efficiency is not close to the ideal 100% efficiency; additional processors further lower the parallel efficiency. To improve inefficiency arising due to unbalanced function evaluations, asynchronous parallel PSO has been proposed. The asynchronous version allows the processors to switch to another function evaluation independently. Performance improvements are obvious for problems with unbalanced and more expensive function evaluations. For example, the parallel efficiency and processor number values now range from (75%, 5) to (90%, 20) [26] and from (77%, 4) to (95%, 32) [41].

As an extension of those efforts in PSO parallelism, we consider how the use of a GPU can accelerate PSO. GPUs were originally designed for image rendering and targeted for mass marketing in desktop and laptop personal computers. However, their high peak floating point operation performance and low deployment cost have offered an attractive parallel computing platform for...
Optimization Methods & Software

scientific and engineering computing. In this article, we focus on using NVIDIA’s GPUs with the compute unified device architecture (CUDA) [33] programming model to solve the optimization problem defined in (1) by PSO.

Several researches in GPU-based PSO have been reported recently. By the means of texture rendering, a PSO is performed on GPU chips [28] whose architecture is different from the programmable general purpose GPU with CUDA. A variation of PSO using digital pheromones is accelerated by using commodity graphics hardware [22]. A parallel approach running the so-called standard PSO [4] is proposed on a CUDA platform [44]. They use another parallelization design that is different from those proposed in this article. GPU versus CPU speedups ranging from around 4 to 11 times are reported. Another CUDA-based PSO implementation is reported in [31]. The number of threads invoked in their approach is equal to the number of particles times the problem dimension, which may limit the dimension of solvable problems. Their algorithm is tested on the Rastrigin function by using three particles. They report up to 50 times speedup. Note that the GPU parallel approaches proposed in [31,44] may be slowed if the objective function contains coupled terms (e.g. $x_1 x_2$) as communications among threads are needed. A hybrid algorithm that combines PSO and pattern search on a GPU is proposed in [45]. Numerical results show that the PSO and pattern search combination can lead to better solutions than PSO only or pattern search only approaches. The speedups of the PSO component are around 5X and the speedups for the pattern search component range from 60X to 200X.

The main purpose is to develop algorithms that use GPUs to accelerate PSO search procedures, especially for higher dimension problems with a large number of particles. The key point for achieving efficiency requires taking advantage of the many-core multiprocessors and the optimized memory bandwidth on the GPU. Numerical results of an implementation of the proposed algorithm on a NVIDIA Tesla C1060 GPU system suggest that the PSO framework lends itself to a GPU implementation. The GPU-accelerated PSO (GPSO) can reduce computational timing, achieve high parallel efficiency, and use a large number of particles to find better optimal solutions. Some numerical findings are highlighted as follows.

- Floating point operations and speedup performance are promising. For example, in a 100-dimensional test problem that is solved by using 65,536 particles and 1000 iterations, we have observed the following. (i) The GPSO achieves 266 giga floating point operations per second (GFLOPS). (ii) There are 280X (5.4 versus 1525.0 s) and 83X (5.4 versus 451.7 s) speedups on a 1.30 GHz GPU against a 3.00 GHz quad-core CPU with a sequential and a parallel implementation, respectively.
- Acceleration results are independent of large numbers of particles. This observation suggests that the proposed algorithm is a promising solver for high-dimensional and complicated problems that need a great quantity of particles.

The remainder of this paper is organized as follows. Section 2 reviews some important PSO algorithms. Section 3 briefly discusses the NVIDIA GPU architecture and environment that are related to our algorithms and implementation. Section 4 discusses our GPSO algorithms and corresponding implementations. Numerical performances in time and computed solutions are discussed in Section 5. Finally, we conclude the paper in Section 6.

2. PSO algorithms

The main concept of PSO is that the position of each individual particle represents one possible candidate solution in the searching space. Each particle updates its velocity and position based on its own experience (cognitive component) and overall swarm experience (social component).
towards a desirable region that has a minimal objective function value. In particular, one step of a common PSO with \( n \) particles is given as the following equation:

\[
v_i = v_i + c_1 \beta_1 (p_i - x_i) + c_2 \beta_2 (p_g - x_i),
\]

\[
x_i = x_i + v_i,
\]

for \( i = 1, \ldots, n \). Here \( v_i \) and \( x_i \) represent velocity and position of the \( i \)th particle, respectively. We let \( p_i \) denote the local best position for the \( i \)th particle. That is, the minimal value of \( f(x) \) that the \( i \)th particle has achieved up to the current step occurs at \( p_i \). We also let \( p_g \) denote the global best position for all particles. That is, the minimal value of \( f(x) \) that all \( n \) particles have achieved up to the current step occurs at \( p_g \).

Furthermore, \( \beta_1 \) and \( \beta_2 \) represent random numbers used to increase diversity. Both \( \beta_1 \) and \( \beta_2 \) are usually determined by a uniform distribution between 0 and 1. The constants \( c_1 \) and \( c_2 \) represent a cognitive learning factor and social learning factor, respectively. They weight the effects for each particle to move towards its own local best position or overall global best position. Obviously, the new velocity defined in (2a) is a linear combination of the three directions.

In comparison with other stochastic algorithms like the simulated annealing [25] and the tabu [16] algorithm, PSO is a population-based method with information sharing. Other population-based methods with information sharing include the genetic algorithm [17] and ant colony algorithm [11].

Although PSO has been used to solve optimization problems arising in various applications, the algorithm based on (2a) may encounter difficulties (e.g. settling on the local optimal solution). Many researchers have developed other variations of the algorithm to increase its efficiency, robustness, and accuracy. We highlight some variations of PSO as follows.

**Inertia weight.** Shi and Eberhart [38,39] introduced a parameter \( \omega \) that is known as inertia weight to modulate the influence of the former velocity. The new velocity updating equation becomes

\[
v_i = \omega v_i + c_1 \beta_1 (p_i - x_i) + c_2 \beta_2 (p_g - x_i).
\]

The inertia weight \( \omega \) can be a constant or be determined by time-varying linear or nonlinear decreasing functions [7,14,39]. A larger inertia weight facilitates global exploration, whereas a smaller one facilitates local exploitation.

**Velocity updating.** Shi and Eberhart [39] suggested using an appropriate maximum velocity in each dimension to restrict the global search (with or without box constraints) or enhance the local search capacity. Clerc and Kennedy [9] introduced the concept of a constriction factor to control the overall velocity updating and to restrict any explosion of particles violating constraints. They also analysed the trajectory for each particle in the search space. The constriction factor ensures convergent behaviour. Mendes et al. [30] proposed a more general velocity updating form known as the fully informed particle swarm algorithm.

**Learning factor.** One of the most common selections for the cognitive and social learning factors is \( c_1 = c_2 = 2 \) [23]. Ide and Yasuda [21] proposed a modified model for dynamically tuning the learning factors. This model adjusts attractive and repulsive behaviour for each particle dynamically based on the variation of the global and local best positions in a period of iterations.

The following algorithm is intended to be run sequentially on a CPU and thus is called CPU-based particle swarm optimization (CPSO). After introducing the GPU architecture in the next section, we propose two variants of CPSO that use GPUs to accelerate the search procedure.
Algorithm 1  CPU-based particle swarm optimization

(0) All computations are done on the CPU.

(1) Initialize

(1.1) Choose initial positions $x_i$ and velocities $v_i$, for $i = 1, \ldots, n$.

(1.2) Evaluate function values $f(x_i)$.

(1.3) Determine local best positions $p_i = x_i$.

(1.4) Determine global best position $p_g$.

(2) Repeat until stopping criteria are satisfied.

(2.1) Update velocities by (3)

(2.2) Update positions by (2b)

(2.3) Evaluate function values $f(x_i)$.

(2.4) Update local best positions $p_i$.

(2.5) Update global best position $p_g$.

3. GPU architecture

A programmable general purpose GPU is a highly parallel, multi-threaded, many-core processor. Unlike CPUs that devote more transistors to data caching and flow control, GPUs are specialized for highly data-parallel computations with high arithmetic intensity. We concentrate our design and implementation of PSO algorithms on the NVIDIA Tesla GPU unified architecture and CUDA version 2.0 [33]. A Tesla GPU contains multiprocessors (or streaming multiprocessors) and each multiprocessor contains eight-thread processors (or scalar processors) and on-chip memory. A CUDA kernel is invoked from the host CPU. Thread blocks of a kernel grid are distributed to available multiprocessors. The threads of a thread block then execute instructions concurrently on one multiprocessor in single-instruction-multiple-thread (SIMT) mode. Each of the multiprocessors creates, manages, schedules, and executes threads in warps, which are groups of 32 parallel threads. Code development on NVIDIA GPUs is realized via the CUDA programming model and software environment [33]. It contains an extension to the C language, compiler, profiler, debugger, libraries, and runtime driver.

As memory access patterns significantly affect the effective bandwidth of each memory space, it is essential to optimize memory bandwidth by following the right access pattern. We highlight the following memory characteristics and memory access patterns that are closely related to our PSO algorithms and implementations. Further details can be found in [33].

CUDA memory hierarchy. Each thread has a private local memory and registers. Threads within the same block can access and share data through a low-latency shared memory that is visible to all of these threads. All threads can access the same global memory, constant memory, and texture memory. CUDA also assumes that the CPU and GPU maintain their own DRAM host memory and device memory, respectively. Note that the shared memory space and registers reside in the on-chip memory. The global, constant, texture, and local memory spaces reside in the device memory.

Global memory and coalescing. When global memory access is necessary, we should follow the access pattern to achieve coalescing that allows global memory bandwidth to be used with the highest efficiency. When half-warp threads access the same segment of global memory, all of the global memory accesses of these threads can be coalesced simultaneously into one single memory transaction.

Shared memory and avoiding bank conflicts. Accessing shared memory space is much faster than accessing local or global memory. Furthermore, if there is no bank conflict among the threads in a warp, accessing the shared memory is as fast as accessing a register. Data in the
different banks (equally sized memory modules) can be accessed simultaneously. However, if multiple memory requests fall in the same bank, the accesses are serialized reducing the efficiency.

*Shared memory and broadcast.* Shared memory also features a *broadcast* to accelerate memory read requests. By servicing one shared memory read request on a common 32-bit word, a particular word can be read and then simultaneously broadcast to several threads. This broadcast can also reduce bank conflicts.

*Constant memory.* After one memory is read from the constant memory residing in the device memory, the data are cached in the chip on-board constant cache. If all half-warps threads read identical addresses, reading data from a constant cache occurs as rapidly as reading data from a register.

*Registers.* Each thread has its own registers. Accessing a register usually needs no extra clock cycles per instruction. While register read-after-write dependencies and register memory bank conflicts may cause delays, such delays can be avoided by suitably choosing number of active threads per multiprocessor or per block.

The general rules for improving the efficiency of GPU computing, such as optimizing memory transfer, maximizing processor occupancy, and optimizing algorithms for GPUs can be found in [18, 33, 34].

4. **GPU-accelerated algorithm and implementation**

The proposed GPSO uses a thread pool model in which all of the PSO steps are performed on a GPU. There are two main ideas in this algorithm. First, each GPU computing thread is in charge of all of the computations of one particle in the SIMT mode. To achieve high timing performance, the threads need to read, write, and exchange necessary data via efficient memory accesses. Second, we form a thread task pool that contains a much larger number of tasks than the number of GPU thread processors. Due to the functionality of memory access hiding [33], the GPU multiprocessors can consequently be kept busy as the multiprocessors switch from one set of SIMT threads executing memory access to another set of SIMT threads taking from the thread task pool to execute the computations. As shown below, such a thread pool model closely maps the PSO framework to the CUDA programming model. Therefore, the GPU can perform numerous tasks using numerous pieces of data efficiently.

We describe GPSO in Algorithm 2. As emphasized in step (0) of Algorithm 2, one key difference between GPSO and CPSO is that the computations corresponding to each of the particles are executed by a single GPU thread. Note that each step of the algorithm is executed by a separate CUDA kernel. Necessary synchronization can then be achieved between different CUDA kernels. For example, all the threads of all the blocks are synchronized after all the threads finish the operations in steps (1.3) and (2.4). Then we update $p_g$ in steps (1.4) and (2.5) by invoking other CUDA kernels.

**Algorithm 2**  
**GPSO based on a thread pool model**

0. All computations are done on GPU. One computing thread represents one particle and executes the following steps with different data. That is, the $i$th thread executes the following steps for $i = 1, \ldots, n$, where $n$ is the number of particles.

1. Initialize
   (1.1) Choose initial positions $x_i$ and velocities $v_i$.
   (1.2) Evaluate function values $f(x_i)$.
Set local best positions \(p_i = x_i\).

Determine global best position \(p_g\).

Repeat until stopping criteria are satisfied.

Update velocities by (3).

Update positions by (2b).

Evaluate function values \(f(x_i)\).

Update local best positions \(p_i\).

Update global best position \(p_g\).

As an appropriate implementation is critical to achieve efficient GPSO, we explain in a step-by-step manner how the implementation of GPSO can (i) minimize memory transformation costs and (ii) maximize arithmetic intensity.

Data structure. The following scalar variables and all other objective function parameters reside in the constant memory.

\[
\begin{align*}
c_1 &: \text{ cognitive learning factor,} \\
c_2 &: \text{ social learning factor,} \\
n &: \text{ number of particles,} \\
D &: \text{ problem dimension,} \\
\omega_i &: \text{ inertia weight of the } i\text{th particle.}
\end{align*}
\]

The following variables reside in the global memory.

\[
\begin{align*}
X \in \mathbb{R}^{n \times D} &: \text{ current positions of all particles,} \\
V \in \mathbb{R}^{n \times D} &: \text{ current velocities of all particles,} \\
p_i \in \mathbb{R}^{n \times D} &: \text{ positions associated with local best values,} \\
p_g \in \mathbb{R}^{1 \times D} &: \text{ position associated with the global best value.}
\end{align*}
\]

Additionally, function values for the current positions \((n \times 1)\), local best values for the particles \((n \times 1)\), and global best value among the particles \((1 \times 1)\) are also stored in the global memory.

The two-dimensional matrix variables \((X, V, \text{ and } p_i)\) are stored in column major order as shown in Figure 1(a). Since each SIMT thread represents a particle, global memory access of these variables is consequently coalesced as illustrated in Figure 1(b). In particular, the coalescing accesses in Algorithm 2 include

(a) reading and writing of \(x_i\) (stored in \(X\)) and \(v_i\) (stored in \(V\)) in step (1.1),

(b) reading of \(x_i\) in steps (1.2) and (2.3),

(c) reading of \(x_i\) and \(p_i\) (stored in \(p_i\)) and writing of \(x_i\) and \(v_i\) in steps (2.1) and (2.2),

(d) reading \(p_i\) in step (2.4).

Furthermore, the memory access of \(p_g\) can also be optimized by broadcast as shown in Figure 2.

Step (1.1). The \(i\)th thread initializes the positions and velocities as follows:

\[
\begin{align*}
\text{for } d = 1, \ldots, D & \quad x(i, d) = \text{rand}(0, 1)(x_d^{\text{max}} - x_d^{\text{min}}) + x_d^{\text{min}} \\
v(i, d) = \text{rand}(0, 1)(v_d^{\text{max}} - v_d^{\text{min}}) + v_d^{\text{min}}
\end{align*}
\]

Here, rand\((0, 1)\) is a random number uniformly distributed in \([0, 1]\); \(x_d^{\text{min}}, x_d^{\text{max}}, v_d^{\text{min}}, \text{ and } v_d^{\text{max}}\) are predefined bounds of positions and velocities in the \(d\)th dimension.

The constant parameters \(x_d^{\text{min}}, x_d^{\text{max}}, v_d^{\text{min}}, \text{ and } v_d^{\text{max}}\) are stored in the constant memory and can be accessed by each thread by broadcast. Figure 3 illustrates how these constants are broadcast.
Figure 1. (a) The column major order in which the current positions (X), current velocities (V), and local best value positions (\( p_i \)) are stored in the global memory. (b) The data structure results in a coalesced memory access pattern as each thread represents a particle. The solid, dotted, and dashed lines indicate the first, second, and third (coalesced) memory access, respectively.

Figure 2. Memory access of \( p_g \) (stored in the global memory) is optimized via broadcast.

Figure 3. Memory accesses of \( x_{\text{min}}^d, x_{\text{max}}^d, v_{\text{min}}^d, \) and \( v_{\text{max}}^d \) (stored in the constant memory) are optimized via broadcast.
Furthermore, the threads compute $x_i$ and $v_i$ simultaneously and then save the positions and velocities in $X$ and $V$.

Steps (1.2) and (2.3). Function evaluation is a core computation in GPSO. As GPU multiprocessors can (i) perform arithmetic and copy data to the registers simultaneously and (ii) switch between blocks very quickly, we can write codes like the one below to take advantage of memory access hiding to avoid memory latency. Each thread performs the following function evaluations to keep the arithmetic logic units on the GPU busy. Take $f(x) = \sum_{d=1}^{D} \sin(x_d)$ as an example.

\[
\begin{align*}
\text{for } d = 1, \ldots, D \\
\text{tmp} &= x_d \\
\text{rslt} &= \sin(\text{tmp}) + \text{rslt} \\
\end{align*}
\]

While threads in one block are computing rslt, threads in another block may read tmp simultaneously. After the computations are performed, the data for the computation in the next loop are ready in the registers; the multiprocessors can start computing the next rslt almost immediately. Besides, data prefetch can also be applied to accelerate the function evaluations.

Step (1.3). For a thread, this step involves a simple data copy, which is a quick device-to-device copy.

Steps (1.4) and (2.5). While all threads cooperate in searching for the global best position, memory coalescing is achieved by the parallel vector reduction algorithm [32]. This algorithm is modified from the CUBLAS $\text{isamin}$ function.

Steps (2.1) and (2.2). Several efficient memory usage techniques can be applied to update the velocities. First, we copy $p_g$ from the global memory to the shared memory. Therefore, threads may access $p_g$ by broadcast as shown in Figure 2. Second, $c_1$ and $c_2$ are stored in the constant memory, which can be accessed by broadcast. Third, we note that $\omega, \beta_1,$ and $\beta_2$ are computed by each thread whenever necessary.

Step (2.4). Unlike other steps, we may encounter possible branch divergences in this step, as not all of the threads in the same warp may require updating of $p_i$. However, the global memory accesses within a half-warp are still coalesced for devices of compute capability 1.2 and higher. Branch divergences can be avoided if all the threads update the corresponding $p_i$, while some threads may simply write the same value. Note that the number of updates gradually decreases as the particles converge to the optimal points.

Another possible way to accelerate PSO is in a client–server model [37]. The CPU acts as a server that manages the overall searching process. The streaming multiprocessors on the GPU act as clients that are responsible for simultaneous function evaluations. While such a scheme parallelizes the most costly part of CPSO (i.e. function evaluations), the communication overhead between the host memory on the CPU and the device memory on the GPU may significantly downgrade the overall performance. Our numerical experiments suggest that such a client–server model achieves up to 4X speedups for the test problems.

5. Numerical experiments

Numerical performance of CPSO (Algorithm 1) and GPSO (Algorithm 2) are compared. In addition to a sequential implementation, we also parallelize steps (2.1), (2.2), (2.3) and (2.4) of CPSO by OpenMP [6] to take advantage of the parallelism of multicore CPUs. We call this
algorithm CPSOMC. The algorithms are implemented by using the C language and NVIDIA CUDA [33]. CPSO and CPSOMC are compiled by using the Intel C compiler icc version 10.1.015 (64-bit). The GPSO is compiled by using GNU gcc version and NVIDIA CUDA compiler. In all compilations, we use optimization level –O3. Numerical experiments for CPSO and CPSOMC are performed on a HP W6400 workstation equipped with an Intel Xeon-X5450 quad-core CPU at 3.00 GHz and 8 GB main memory. The operating system is Linux Ubuntu 8.04 64-bit server version. The results of GPSO are performed on a CPU–GPU system composed of the HP workstation and an NVIDIA Tesla C1060 GPU card. The card contains 240 thread processors organized in 30 multiprocessors (eight thread processors per multiprocessor), and its clock rate is 1.30 GHz. The card also has 4 GB global memory, 480 kB shared memory (16 kB per multiprocessor), and 64 kB constant memory. Single precision (SP) floating point arithmetic is used in the numerical experiments, unless explicitly indicated. Due to a system upgrade of the test machine, numerical results of Section 5.4 and the ones related to CPSOMC are obtained by using icc version 11.1.072 (64-bit) and Linux Ubuntu 9.04 64-bit server version.

The parametric optimization problems taken from [19] are listed in Table 1. These problems are used as the test problems in the numerical experiments by letting the dimension $D = 100$. Note that the settings of iterations, $D$, $c_1$, $c_2$, and $\omega$ used in our experiments and [19] are the same. We assume that the learning factor $c_1 = c_2 = 2$ and the maximum allowable velocity was the full search space. For the inertia weight function $\omega$, we use a popular heuristic that is linearly decreased [38,39] from 0.9 to 0.3 as suggested by [20]. The three algorithms are terminated after 1000 iterations. The numbers of particles are chosen from $16 \times 2^0$, $16 \times 2^1$, $16 \times 2^2$, …, and so on through $16 \times 2^{16} (= 1,048,576)$. To determine the thread number per block (a hardware-dependent parameter that significantly affects the efficiency of the codes), a simple scan test was run using 32 to 512 threads with increments of 32 threads (warp size). In particular, we choose the block size of 256 based on the tuning results shown in Table 2.

5.1 Timing performance

We investigate the timing performance of CPSO, CPSOMC, and GPSO in terms of both total time and acceleration. We use different numbers of particles and a fixed iteration number 1000. We focus the timing performance analysis on four test functions ($f_1$, $f_2$, $f_3$, and $f_4$), as they representatively demonstrate performance behaviours of all test functions.

The speedups achieved by GPSO compared with CPSOMC and CPSO are shown in Figure 4 where we plot the time ratios

\[
\frac{T_{\text{CPSO}}}{T_{\text{GPSO}}} \quad \text{and} \quad \frac{T_{\text{CPSOMC}}}{T_{\text{GPSO}}}
\]

for a range of particles. Here, $T_{\text{CPSO}}$, $T_{\text{CPSOMC}}$, and $T_{\text{GPSO}}$ denote the total times (in seconds) consumed by CPSO, CPSOMC, and GPSO, respectively. The figures show that the accelerations of GPSO versus CPSO increase exponentially with number of particles and reach maxima of 281.7X, 257.8X, 104.5X, 191.8X while minimizing functions $f_1$, $f_2$, $f_3$, and $f_4$, respectively. On the other hand, the accelerations of GPSO versus CPSOMC reach maxima of 83.4X, 79.5X, 31.5X, and 59.0X for the first four test functions. The accelerations are sustained at their highest levels as the number of particles increase further.

Another way to evaluate the performance of GPSO is to measure its floating point operations. Figure 5(a) shows GFLOPS achieved by GPSO averaged over 30 runs for solving the problem $f_1$. The floating point operations are counted by the CUDA Visual Profiler. GPSO achieves maximum performance of 266 GFLOPS for 65,536 particles, and then stabilizes around 250 GFLOPS. Clearly, the GFLOPS curve in Figure 5(a) is similar to the speedup curve of function $f_1$ shown in Figure 4(a). GPSO keeps the multiprocessors busy almost all the time, uses efficient memory
Table 1. Parametric optimization test problems used in [19].

<table>
<thead>
<tr>
<th>Function</th>
<th>$x_i$ Domain</th>
<th>Optimum</th>
<th>Function name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1 = \sum_{i=1}^{D} [\sin(x_i) + \sin(2x_i/3)]$</td>
<td>[3, 13]</td>
<td>$-1.21598D$</td>
<td>[27]</td>
</tr>
<tr>
<td>$f_2 = \sum_{i=1}^{D-1} [\sin(x_i + x_{i+1}) + \sin(2x_i x_{i+1}/3)]$</td>
<td>[3, 13]</td>
<td>$\approx -2(D - 1)$</td>
<td>Krishna Kumar</td>
</tr>
<tr>
<td>$f_3 = \sum_{i=1}^{D} [x_i + 0.5]^2$</td>
<td>$[-100, 100]$</td>
<td>0</td>
<td>[19]</td>
</tr>
<tr>
<td>$f_4 = \sum_{i=1}^{D} [x_i^2 - 10 \cos(2\pi x_i) + 10]$</td>
<td>$[-5.12, 5.12]$</td>
<td>0</td>
<td>Rastrigin</td>
</tr>
<tr>
<td>$f_5 = \sum_{i=1}^{D} x_i^2$</td>
<td>$[-5.12, 5.12]$</td>
<td>0</td>
<td>Sphere</td>
</tr>
<tr>
<td>$f_6 = \sum_{i=1}^{D} x_i \sin(10\pi x_i)$</td>
<td>$[-1.0, 2.0]$</td>
<td>$-1.85D$</td>
<td>–</td>
</tr>
<tr>
<td>$f_7 = \sum_{i=1}^{D}</td>
<td>(\sin(10\pi x_i))/10\pi x_i</td>
<td>$</td>
<td>$[-0.5, 0.5]$</td>
</tr>
<tr>
<td>$f_8 = 20 + e - 20e^{-0.2\sqrt{(\sum_{i=3}^{D} x_i^2)/D}} - e^{\sum_{i=3}^{D} (\cos(2\pi x_i))/D}$</td>
<td>$[-30, 30]$</td>
<td>0</td>
<td>Ackley</td>
</tr>
<tr>
<td>$f_9 = 418.9829D - \sum_{i=1}^{D} x_i \sin(\sqrt{</td>
<td>x_i</td>
<td>})$</td>
<td>$[-500, 500]$</td>
</tr>
<tr>
<td>$f_{10} = \sum_{i=1}^{D-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$</td>
<td>$[-5.12, 5.12]$</td>
<td>0</td>
<td>Rosenbrock</td>
</tr>
<tr>
<td>$f_{11} = 6D + \sum_{i=1}^{D}</td>
<td>x_i</td>
<td>$</td>
<td>$[-5.12, 5.12]$</td>
</tr>
<tr>
<td>$f_{12} = \frac{1}{4000} \sum_{i=1}^{D} x_i^2 - \prod_{i=1}^{D} (x_i/\sqrt{i}) + 1$</td>
<td>$[-600, 600]$</td>
<td>0</td>
<td>Griewank</td>
</tr>
</tbody>
</table>

The problem dimension $D$ is chosen to be 100 in the numerical experiments.

Table 2. Thread numbers per block and the corresponding average time (out of 30 independent runs) in seconds for solving problem $f_1$ by using 1,048,576 particles and 1000 iterations.

<table>
<thead>
<tr>
<th>Thread no.</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>131.872</td>
<td>124.986</td>
<td>112.781</td>
<td>107.013</td>
<td>107.237</td>
<td>106.621</td>
<td>106.892</td>
<td>106.371</td>
</tr>
</tbody>
</table>

| Thread no. | 288 | 320 | 352 | 384 | 416 | 448 | 480 | 512 |
| Time (s)   | 106.912 | 106.621 | 106.763 | 107.927 | 113.253 | 120.912 | 127.289 | 142.908 |

Since a warp contains 32 threads, the block sizes are chosen as multiples of 32.

access patterns, and hides the memory accesses and thus results in the efficient floating point operations.

We next undertake deeper investigations regarding the behaviour of GPSO by answering the following questions.

- Why does the acceleration slightly decline after achieving the top performance as shown in Figure 4?
Figure 4. Speedups achieved by GPSO for test functions $f_1$, $f_2$, $f_3$, and $f_4$. The time ratios $(T_{CPSO}/T_{GPSO})$ and $(T_{CPSOMC}/T_{GPSO})$ are illustrated for different numbers of particles. $T_{CPSO}$, $T_{CPSOMC}$, and $T_{GPSO}$ are the total times consumed by the algorithms CPSO, CPSOMC, and GPSO, respectively. The number of algorithm iterations is equal to 1000.

Figure 5. GFLOPS performance of GPSO versus number of particles for test functions $f_1$, $f_2$, $f_3$, and $f_4$. The number of iterations is equal to 1000.

This phenomenon is likely to be due to the additional memory access bandwidth requirements in local best updates (i.e. step (2.4) in Algorithm 2), especially after full GPU workload has been achieved.
Figure 6. Timing performance of GPSO versus number of particles for 1000 iterations. The total time roughly equals the kernel time (i.e. the function evaluation time) plus the update and communication times. The update time is almost function-independent.

- Why are there different levels of acceleration?
  While the accelerations for test problems \( f_1 \) and \( f_2 \) reaches \( 270X \), \( f_3 \) reaches about \( 100X \). As shown in Figure 6, GPSO actually takes a similar total amount of time for different test problems. However, CPSO spends more time on solving problems \( f_1 \) and \( f_2 \); therefore, GPSO demonstrates a greater acceleration. The slowing in CPSO is mainly due to the function evaluations. Functions \( f_1 \) and \( f_2 \) are more complicated and involve sine functions. In contrast, \( f_3 \) requires only squares.

- What are the relationships between acceleration and function evaluation costs?
  It is worth mentioning that the function evaluations affect performance of both CPUs and GPUs in a similar manner. Evaluations for \( f_1 \) and \( f_2 \) are slower than \( f_3 \) (see the plots of kernel time for function evaluations in Figure 6). However, the GPU is more efficient in floating point arithmetic. Therefore, the kernel time contributes little to the total time. In contrast, the function evaluation time spent on the CPU affects the total time more significantly. Consequently, in a fixed update and communication cost scenario, an expensive function results in better acceleration than does a cheap function.

- Why are there turning points in the lines of Figure 6?
  The GPSO timing results remain almost constant when the number of particles is less than \( 10^4 \). The timing results increase linearly thereafter. The constant-like speedup occurs because of the smaller number of particles induced low GPU workload. The turning points also indicate the critical number of particles for better taking advantage of the GPU workload.

For the timing measurements for CPSO and CPSOMC, we (i) perform loop unrolling, (ii) design data structure for the cache and main memory to be used in an efficient way, and (iii) use single precision sine and cosine in function evaluations. For the test functions that involve sines and cosines, the GPU can take advantage of special function units that are embedded in the multiprocessors for efficient trigonometric function evaluations [33]. However, we choose not to use these units in order to make a fair comparison against the CPU.
5.2 Solution quality of the computed minima

Table 3 presents the average, best, and worst results of 30 independent runs using $16 \times 2^{16}$ particles with 1000 iterations by GPSO. The optimal solutions are listed in the last column of the table. The table suggests that GPSO solves the problems $f_1$, $f_3$, $f_5$, $f_8$, and $f_{12}$ to the desired accuracy and does not solve adequately the other problems and obtains solutions that are not so well or still far away from the optima for other functions. Such results are expected as many of the test problems are non-trivial in the sense that many of these test functions have a massive number of local minima. These local minima increase the degree of difficulty for finding the global minimum. Test functions $f_1$–$f_4$ are plotted in Figure 7 for $D = 2$. Besides, some of these test problems are discontinuous and some are not smooth. Computational results of other variants of PSO for the same test functions can be found in [19].

5.3 Effect of particles

Figure 8 shows the number of particles versus function values (computed by averaging 30 runs) after 1000 iterations for test functions $f_1$, $f_2$, $f_3$, and $f_4$ defined in Table 1. The results suggest that the number of particles affects the convergence rate in two ways. In the Type-I problems (e.g. $f_1$ and $f_3$), the convergence rate increases almost exponentially with respect to the number of particles in the interval $[16, 16 \times 2^{16}]$. Particles that more than $16 \times 2^6$ in $f_1$ or $16 \times 2^4$ in $f_3$ contribute little to the convergence rate of the function values. In contrast, in the Type-II problems (e.g. $f_2$ and $f_4$), the convergence rate is almost linear with respect to the number of particles in the whole interval $[16, 16 \times 2^{16}]$. The test functions, $f_5$, $f_8$, $f_{10}$, and $f_{12}$ are of Type-I, and $f_6$, $f_7$, $f_9$, and $f_{11}$ are of Type-II. Consequently, increasing the number of particles increases the search speed of PSO.

5.4 Effect of computer arithmetic

SP and double precision (DP) arithmetic can affect CPSO and GPSO in terms of timing performance and solution quality. Usage of different computer arithmetic precision can result in a trade-off between timing efficiency and solution accuracy. Such a trade-off on GPU in numerical linear algebra are studied in [12,40]. Computations based on SP (DP) arithmetic have higher (lower) GFLOPS, but they usually lead to lower (higher) solution accuracy. Note that

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Best</th>
<th>Worst</th>
<th>Exact</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>-121.598</td>
<td>-121.598</td>
<td>-121.598</td>
<td>-121.598</td>
</tr>
<tr>
<td>$f_2$</td>
<td>-170.073</td>
<td>-173.232</td>
<td>-164.705</td>
<td>-200.000</td>
</tr>
<tr>
<td>$f_3$</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>$f_4$</td>
<td>66.701</td>
<td>50.643</td>
<td>87.219</td>
<td>0.000</td>
</tr>
<tr>
<td>$f_5$</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>$f_6$</td>
<td>-98.141</td>
<td>-108.593</td>
<td>-91.611</td>
<td>-185.000</td>
</tr>
<tr>
<td>$f_7$</td>
<td>0.192</td>
<td>0.004</td>
<td>0.942</td>
<td>0.000</td>
</tr>
<tr>
<td>$f_8$</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>$f_9$</td>
<td>13,697.234</td>
<td>12,659.520</td>
<td>14,883.352</td>
<td>0.000</td>
</tr>
<tr>
<td>$f_{10}$</td>
<td>127.815</td>
<td>84.218</td>
<td>252.997</td>
<td>0.000</td>
</tr>
<tr>
<td>$f_{11}$</td>
<td>39.600</td>
<td>22.000</td>
<td>66.000</td>
<td>0.000</td>
</tr>
<tr>
<td>$f_{12}$</td>
<td>0.002</td>
<td>0.000</td>
<td>0.010</td>
<td>0.000</td>
</tr>
</tbody>
</table>

The average, best, and worst results of 30 independent runs for each function are reported. The iteration numbers of GPSO are set to 1000 with $16 \times 2^{16}$ particles.
the theoretical GFLOPS ratio of SP versus DP is about 2:1 on a CPU and about 8:1 on the NVIDIA Tesla C1060 GPU that has 240 SP and 30 DP scalar processors.

We demonstrate results for solving problem $f_2$ by using CPSO and GPSO with $16 \times 2^{16}$ particles and 1000 iterations. For other test problems with various particle and iterations settings, we have observed similar behaviour and thus we skip the details. Figure 9(a) shows the average function values of the 30 independent trials for all iterations. The differences between SP and DP are insignificant overall, especially in the first 850 iterations. In these early and middle stages, the number and locations of the points that have been explored have larger effects than the accuracy of function evaluations (related to arithmetic precision). In the stage from 850 to 1000 iterations, CPSO and GPSO gradually converge to the optimal solutions as shown in Figure 9(b). In this stage, the DP results are more accurate due to the particles gathering around the optimal point.
This increases the accuracy of the global best function evaluations which usually increases even more the solution accuracy.

Figure 10 shows the average function values and average time in seconds. The figure suggests that all four methods converge to very similar function values: $-169.3023217$ (GPSO SP),
Figure 10. Values of test function $f_2$ versus time (computed by averaging 30 independent runs) for 1000 iterations. $;SP$ and DP stands for single precision and double precision, respectively.

However, GPSO is about 220$X$ and 58$X$ faster than CPSO when SP and DP arithmetic is used, respectively. In particular, the time values in seconds are $8.190 \times 10^1$ (GPSO SP), $4.123 \times 10^2$ (GPSO DP), $1.812 \times 10^4$ (CPSO SP), and $2.412 \times 10^4$ (CPSO DP).

These experiments lead to the following remarks.

- We can use a mixed-precision scheme to balance efficiency and accuracy. That is, initially we may use GPSO to get an efficient global search, until the improvement of function values are small. Then we shrink the search domain and switch to GPSO DP to conduct a local search to get higher accuracy solutions.
- Both CPSO and GPSO share the same underlying PSO framework. Thus, we can expect that CPSO and GPSO converge in a similar way in the average sense, as shown in Figure 9(a). In other words, by setting a fixed accuracy requirement, CPSO and GPSO will use a similar number of iterations to achieve the desired accuracy. The corresponding timing performance speedups will be similar to those shown in Figure 4.

6. Conclusion

In the study of box-constrained minimization problems, we have proposed GPSO (Algorithm 2) that uses the NVIDIA Tesla C1060 GPU to improve the timing efficiency of PSO. We have also provided detailed explanations of the implementation techniques to show how we took full advantage of the numerous computational threads with appropriate memory access. Numerical
results show that the target GPU hardware model can be very efficient for the PSO algorithm, and GPSO may significantly reduce the total running time and retain a highly parallel efficiency for a large number of particles.

The efficiency of GPSO can be affected by the target optimization problems. As the test problems have many local minima, population-based algorithms, such as PSO, usually have better chances to find the global optimal or better sub-optimal solutions than local search methods like steepest descent. While we have demonstrated that GPU can be an efficient accelerator to PSO for solving these test functions, other stochastic algorithms like genetic algorithms [17], simulated annealing [25], tabu [16], and ant colony [11] can be tailored to fit on a GPU. On the other hand, unlike the test problems considered here, some objective functions are determined by a software package. It can be a non-trivial task to implement the whole software package on a GPU. Consequently, a PSO over a CPU cluster or a CPU-and-GPU hybrid cluster would be a reasonable choice. Of course, the relative cost of the function evaluation and data transfer overhead between computing nodes of the cluster can affect the speedups.

We believe that the advantages of using a GPU to accelerate PSO are not limited to the results reported in this article. In addition to porting other variants of PSO to GPU, we may apply ideas like orthogonal designs for choosing variables and the integration of global and local searches to further improve the proposed algorithms. We may also use a mixed-precision technique that uses low-precision arithmetic in the initial stage and then uses high-precision arithmetic to do an accurate local search. It is also possible to combine GPSO with pattern search to gain better solution quality as suggested in [45]. We anticipate such approaches to be suitable for a GPU, as both GPSO and pattern search map efficiently on a GPU as shown in this article and in [45], respectively. These future studies in algorithms and implementations have great potential to allow efficient PSO to tackle very high-dimensional problems.

Acknowledgements

The authors are grateful to the anonymous referees and Dianne P. O’Leary for helpful comments and suggestions. This work is partially supported by the National Science Council, the Taida Institute for Mathematical Sciences, and the Mathematics Division of the National Centre for Theoretical Sciences (Taipei Office) in Taiwan.

References

Optimization Methods & Software


