A Survey on GPU-Based Implementation of Swarm Intelligence Algorithms

Ying Tan, *Senior Member, IEEE*, and Ke Ding

***Abstract*—Inspired by the collective behavior of natural swarm, swarm intelligence algorithms (SIAs) have been developed and widely used for solving optimization problems. When applied to complex problems, a large number of fitness function evaluations are needed to obtain an acceptable solution. To tackle this vital issue, graphical processing units (GPUs) have been used to accelerate the optimization procedure of SIAs. Thanks to their inherent parallelism, SIAs are very suitable for parallel implementation under the GPU platform which have achieved a great success in recent years. This paper presents a comprehensive review of GPU-based parallel SIAs in accordance with a newly proposed taxonomy. Critical concerns for the efficient parallel implementation of SIAs are also described in detail. Moreover, novel criteria are also proposed to evaluate and compare the parallel implementation and algorithm performance universally. The rationality and practicability of the proposed optimization methodology and criteria are verified by careful case study. Finally, our opinions and perspectives on the trends and prospects on the relatively new research domain are also presented for future development.**

***Index Terms*—Compute unified device architecture (CUDA), GPU computing, heterogeneous computing, open computing language (OpenCL), population-based algorithms, swarm intelligence algorithms (SIAs).**

I. Introduction

SWARM intelligence (SI) is the collective behavior of decentralized and self-organized systems. A typical SI system consists of a population of simple agents which can communicate with each other by acting on their local environments. Though the agents in a swarm follow very simple rules, the interactions between such agents can lead to the emergence of very complicated global behavior, far beyond the capability of single individual agent [1], [3], [4]. Examples in natural systems of SI include bird flocking, ant foraging, fish schooling, just to name a few.

Inspired by such behavior of a swarm, a class of algorithms have been proposed for tackling optimization problems, usually under the title of SI algorithms (SIAs) [2], [5]. In SIAs, a swarm is made up of multiple artificial agents. The agents can exchange heuristic information in the form of local interaction. Such interaction, in addition to certain stochastic elements, generates the behavior of adaptive search, and finally leads to global optimization.

The most respected and popular SIAs are particle swarm optimization (PSO) which is inspired by the social behavior of bird flocking or fish schooling [6], [7], and ant colony optimization (ACO) which simulates the foraging behavior of ant colony [8], [9]. PSO is widely used for real-parameter optimization while ACO has been successfully applied to solve combinatorial optimization problems. Very recently, inspired by explosion of fireworks in air, a novel SIA, so-called fireworks algorithm (FWA) was proposed in 2010 for efficiently solving complex optimization problems and had received extensive attention [2], [3], [10].

Although SIAs achieve success in solving many real-world problems where conventional arithmetic and numerical methods fail, they suffer from the drawback of intensive computation which greatly limits their applications where function evaluation is time-consuming.

Facing technical challenges with higher clock speeds in fixed power envelope, modern computer systems increasingly depend on adding multiple cores to improve performance [11], [12]. The multicore revolution, along with other factors, encourages the community to start looking at heterogeneous solutions: systems which are assembled from different subsystems, each of which is optimized for different workload scenarios [13]. Nowadays, all computing systems, from mobile to supercomputers, are becoming heterogeneous, massively parallel computers for higher power efficiency and computation throughput [14].

Heterogeneous computing, which refers to systems that use more than one kind of processor, has entered computing's mainstream. Systems with diverse heterogeneous combinations have been applied in the scientific domain [15]. General purpose computing on the graphical processing unit (GPGPU) is one of most important the heterogeneous solutions.

Initially designed for addressing highly computational graphics tasks, the graphical processing unit (GPU), from its inception, has many computational cores and can provide massive parallelism (with thousands of cores) at a reasonable price. As the hardware and software for GPU programming grow mature [16], [17], GPUs have become popular for general purpose computing beyond the field of graphics processing, and great success has been achieved in

Manuscript received February 11, 2015; revised April 21, 2015; accepted July 21, 2015. Date of publication November 10, 2015; date of current version August 16, 2016. This work was supported in part by the Natural Science Foundation of China under Grant 61375119, Grant 61170057, and Grant 60875080, and in part by the National Key Basic Research Development Plan (973 Plan) Project of China under Grant 2015CB352302. This paper was recommended by Associate Editor P. P. Angelov.

The authors are with the Key Laboratory of Machine Perception, Department of Machine Intelligence, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China (e-mail: ytan@pku.edu.cn; keding@pku.edu.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCYB.2015.2460261

diverse domains, from embedded systems to high performance supercomputers [18]–[20].

Based on interactions within population, SIAs are naturally amenable to parallelism. SIAs' such intrinsic property makes them very suitable to run on the GPU in parallel, thus gain remarkable performance improvement.

Although the attempt on leveraging GPUs' massively parallel computing power can date back to the first day of GPGPU [21], [22], significant progress had not been achieved until the emergence of high-level programming platforms such as compute unified device architecture (CUDA) and open computing language (OpenCL) [23], [24]. In the past few years, different implementations of diverse SIAs were proposed. Targeting on GPUs of various architectures and specifications, many techniques and tricks were introduced. Some variants were designed specifically to suit for GPUs particular architecture such as [25]–[27].

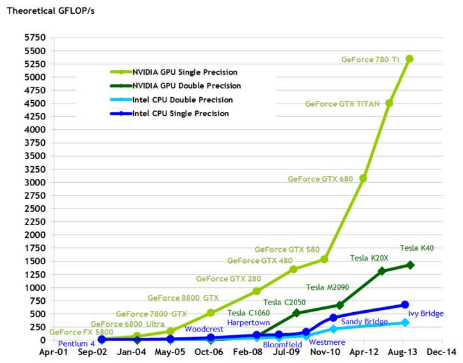
With the excellence in performance, GPU-powered SIAs have been applied to many complex real-world problems, such as image processing [28]–[30], computer vision [31]–[34], machine learning [35]–[37], data mining [38], [39], parameter optimization [40]–[42], economy [43]–[45] as well as many other problems (see [46]–[49]). Thanks to the significant speedup, SIAs now can be used to solve many tasks which are previously unmanageable by the original algorithm in a reasonable time.

Although much effort has been made, research of GPU-based SIAs is still in its infancy and rising phase, especially good and convincible performance criteria are yet to be developed. In order to better understand what achievements have been made and obtain a useful insight on the future development, we perform an exhaustive literature review of the published works in this field. Hopefully, lights can be shed on the trends of SIAs on the GPU, a relatively new member of general-purpose hardware platform.

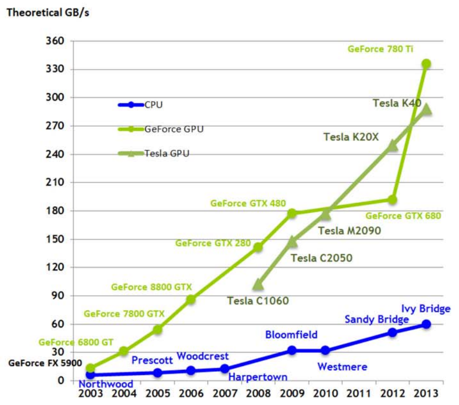
The remainder of this paper is organized as follows. Section II gives a brief introduction of general-purpose computing based on GPU. In Section III, a novel taxonomy is proposed and established, then used to review literature in detail. The subsequent section is dedicated to the detailed parallelization implementation techniques, including efficient parallel algorithm and refining memory access. In Section V, we discuss the evaluation criterion used for measuring the benefit of parallelization and comparing different implementations and present our proposed criteria for that purpose. The verification of the proposed criteria will be presented in Section VI-B1 with detailed case study. Finally, this paper is concluded with our insights and opinions on the trends and perspectives in the field of GPU-based SIAs.

II. General-Purpose Computing on GPUs

Driven by the insatiable demand for real-time high-definition graphics, GPUs have evolved into highly parallel, many-core processors and are able to execute tens of hundreds threads concurrently. Today's GPUs greatly outperform CPUs in both arithmetic throughput and memory bandwidth (see Fig. 1). GPUs can offer great performance



(a)



(b)

Fig. 1. GPUs greatly outperform CPUs in both arithmetic throughput and memory bandwidth [52]. (a) Floating-point operations per second for the CPU and GPU. (b) Memory bandwidth for the CPU and GPU.

at a very low price, and meanwhile GPUs can also be integrated into high performance computing systems without much difficulty [50], [51]. Moreover, GPUs also have great performance/watt, which is key for achieving super computing performance. In the latest (as of April 2015) Green500 list,[1] nine of the top ten systems on the Green500 are accelerated with GPUs. Much effort has been made to harness the enormous power of GPUs for general-purpose computing, and a great success has been achieved.

Many platforms and programming models have been proposed for GPU computing, of which the most important platforms are CUDA [52] and OpenCL [53]. Both platforms are based on C language and share very similar platform model, execution model, memory model and programming model.

CUDA, a proprietary architecture from NVIDIA, enjoys the advantage of mature ecosystem and it is very easy to use as far as programming procedure is concerned. CUDA is just like conventional C/C++ except adding a few of GPU platform specific key words and syntax. Also, there exist interfaces with other high-level languages. With pyCUDA and Jacket toolbox, programmer can accelerate Python and MATLAB Codes utilizing CUDA under the hood, respectively. By now, CUDA is a dominant platform for GPU-targeted SIAs as well as GPGPU.

[1]http://green500.org/lists/green201411

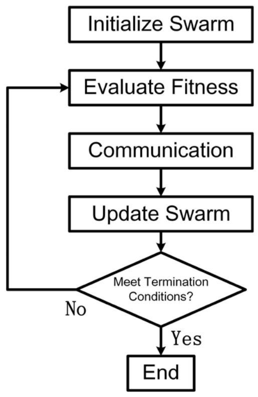


Fig. 2. Framework of SIAs.

OpenCL is an open, royal-free specification aimed at heterogeneous programming platforms. OpenCL is portable and supported by diverse computing devices, including GPUs (both NVIDIA and AMD), CPUs, accelerators (e.g., Intel Phi), FPGA, and so forth [15]. As smart phones, tablets, and wearable devices are increasingly popular, there are good chances that OpenCL will be used on these novel embedded computing devices [54]–[56].

Besides OpenCL and CUDA, other easy-to-use programming tools are also available for leveraging the computational power of GPUs, e.g., OpenACC and C++ accelerated massive parallelism. A detailed description on GPGPU is out of the scope of this paper. For a systematic and completed knowledge on GPU computing, readers can refer to [17] and [57].

III. Swarm Intelligence Algorithms Based on GPUs

In essence, SIAs are iterative-based stochastic search algorithms where heuristic information is shared in order to guide the search in the following iterations. A simplified general framework of SIAs is depicted in Fig. 2. For a particular SI algorithm, the sequence of each phase may be different and some phases can be included several times in a single iteration.

For a particular SIA, each phase is with parallelism of various degree, thus should be mapped onto the most suitable processors (CPU or GPU) for optimal performance. In accordance with how each phase is mapped onto processors (the CPU or the GPU), the GPU-based implementations can be categorized into four major categories.

1) Naive parallel model.
2) Multiphase parallel model.
3) All-GPU parallel model.
4) Multiswarm parallel model.

A. Comparison With Related Work

Before starting the literature survey by the proposed taxonomy, we take a brief comparative review of the related work in literature and show how our proposal differs from the previous ones.

Krömer *et al.* [58] provided a brief overview of the research on the design, implementation, and applications of GPU-based PSO. Works published in 2012 and 2013 were reviewed chronologically, and each proposal was described independently.

Augusto *et al.* [59] presented and discussed different parallelization strategies for implementing ACO on GPU. Special attention was given to OpenCL by the authors.

Extending the work in [60], Krömer *et al.* [61] provided a brief survey of recent studies dealing with GPU-powered genetic algorithm (GA), differential evolution, PSO, and simulated annealing (SA) as well as the applications of these algorithms to both research and real-world problems. In this review, each group of algorithms were independently reviewed instead of under a uniform framework.

Valdez *et al.* [62] implemented on the GPU a set of bio-inspired optimization methods (PSO, GA, SA, and pattern search). However, implementation details were not described. Reviewed several evolutionary computation (EC) methods, respectively, Majd and Sahebi [63] summarized four general frameworks for the parallelization of EC methods. However, no real implementation or specific hardware was involved.

Though these previous works are helpful for our understanding of GPU-based SIAs in particular way, they have their own shortcomings. None of them can serve as a framework which both enables us studying GPU-based SIAs in a universally way and guides the hands-on implementation of SIAs on the GPU. In [61] and [62], no framework was built to discuss different SIAs and related algorithms. While some framework was used in [59] and [63], respectively, the former proposal targets only on GPU-based ACO and the later one provides no discussions about implementation. Different implementations were discussed in [58], however, no discussion about how these implementations are related to each other and what are their pros and cons. The four proposed taxonomies are mainly aiming at CPU-based parallel platforms [60], which is not suitable for GPU-based SI algorithms.

Inspired by all those previous works, we come up with a new taxonomy for parallel implementation of SIAs based on the principals of parallel optimization [64]. With the proposed taxonomy, GPU-based implementation of diverse SIAs can be studied and compared under a uniform framework. Besides a high-level taxonomy, critical implementation considerations are presented in sufficient detail for the purpose of real-world implementation. We think the proposed taxonomy will be helpful for those who want to leverage GPUs' enormous parallel computing power for accelerating SIAs of specific kind as well as more practical for real-world applications.

With the taxonomy, SIAs of different kinds are classified and surveyed extensively in the remainder of this section.

B. Naive Parallel Model

SIAs are suitable for black-box optimization problems, i.e., the algorithms need not to know the structure of the objective function, only require that giving a trail solution, and the corresponding fitness value (cost) can be returned. For nontrivial problems, evaluating a fitness function for every agent is usually very time-consuming. Typically, each fitness is evaluated in a parallel, independent way. Thus, according to the Amdahl's Law [65], significant speedup can be obtained by

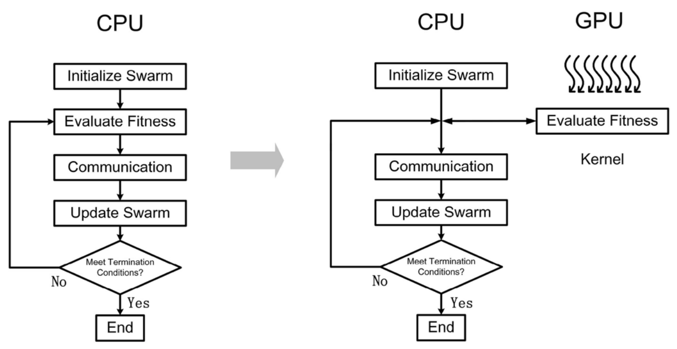
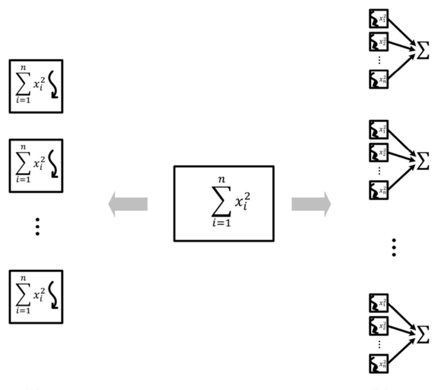


Fig. 3. Naive parallel model.



(a) (b)

Fig. 4. Example for (a) coarse-grained and (b) fine-grained parallel.

parallelized fitness evaluation. By a breakdown timing of each part of the algorithm, the potential speedup can be derived easily before real-parallelization begins.

So by considering the basic principle in parallel programming—locate the bottleneck and optimize it, the straightforward optimization strategy to accelerate SIAs is to offload the fitness evaluations onto the GPU for paralleling running. Compared to other more complicated strategies (as will be discussed later) which can utilize more parallelism in specific algorithms, such an implementation is relatively naive and can be used by almost any SIA, so we name it naive parallel model as illustrated in Fig. 3.

Implementation using naive parallel model can be coarse-grained (task parallel) or fine-grained (data parallel).

When fitness function lacks parallelism or there are many independent evaluations to be performed simultaneously, the coarse-grained parallel can be adopted. Under this condition, each evaluation task is mapped onto one thread [see Fig. 4(a)].

Naive parallel model is particularly interesting when the fitness function can be parallelized. In that case, the function can be viewed as an aggregation of a certain number of partial functions that can be run in parallel, i.e., a fine-grained implementation [see Fig. 4(b)]. Many implementations for optimizing benchmark functions fall into this category [66], [67], so do many real-world applications [68], [69].

Naive as this parallel model is, it can be very useful in both academic research and industrial applications. As only fitness evaluation is selected for parallel implementation, legacy

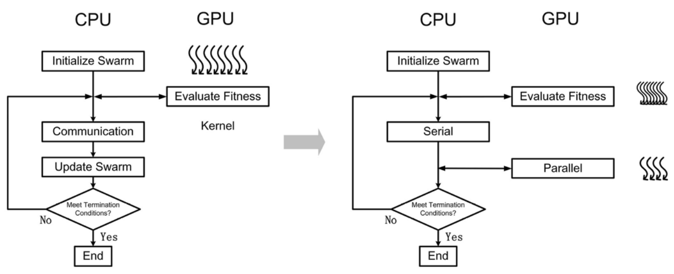


Fig. 5. Multiphase parallel model.

serial codes need little change, thus simplify the programming and debugging process while greatly reducing the running time of the program. Besides, the parallelized fitness evaluation component is "pluggable." Researchers can implement different SIAs without worrying too much about the performance, and plug the GPU-parallelized fitness evaluation for fast execution. This can greatly accelerate the process of comparing different algorithms and developing novel algorithms. For this purpose, a GPU-based benchmark has been proposed and reported in [70].

C. Multiphase Parallel Model

Naive parallel model offers a useful starting point when considering GPU-based parallelization of SIAs, and is popular due to its easy implementation. However, an acceptable acceleration may not be obtained with this model.

Once the explicit parallel part is parallelized and highly optimized, the percentage of the serial part of the program (scaled serial ratio) will increase correspondingly [71]. In these cases, further exploiting the parallelism of the remainder parts (e.g., velocity and position update in PSO and more generally finding the top n maximum/minimum values, sorting, etc.) can be beneficial. With reasonable design, more efficient implementation is possible, thus able to fully leverage GPUs computing power.

In the multiphase parallel model, attempts are made to offload computation from different phases with explicit or implicit parallelism onto GPUs (see Fig. 5).

Unlike the case of naive parallel model which is universal for all SIAs, multiphase parallel model is algorithm dependent. Multiple kernels can be implemented for parallelizing different phases, and different kernels can take different parallel strategies and granularity to best fit to the specific task.

1) Vector Operations: Many operations in SIAs can be formulated as vector operations, such as velocity and position update in PSO [23], movement and feeding operators in Fish School Search (FSS) [72] and spark generation in FWA [10], [25]. Usually, vector operations can be implemented in a very fine-grained data parallel way, i.e., each entry of the vector is updated by a single thread. Such a technique is very straightforward and trivial, so adopted by all SIA implementations reviewed.

2) Reduction: Reduction is a core building block for parallel computing. Finding minimal or maximal values in FWA [10], [25], calculate the sum in FSS [72], and distance calculation in firefly algorithm and some PSO variants [73]–[75] are all instances of reduction primitive. For

GPUs of older architectures (fermi or older), reductions can be implemented efficiently using share memory and atomic operators. For Kepler or newer GPUs, more effective implementations are available via the register-based shuffle operator [52].

3) Sorting: Sorting is a basic block for many SIAs, such as Bees algorithm [26] and Cuckoo algorithm [27]. Though not the best for the GPU, it may be desirable to have a GPU implementation especially when data is already resident in the GPU. Efficient algorithms for GPU sorting can be found in [76]. For Kepler or newer GPUs, more efficient implementations have been proposed in [77], where an alternative sorting enactor designed specifically for small problem sizes is described. In some cases, reduction can be utilized to replace partially sorting, which functions similarly with respect to the final results [27].

4) Path Construction: One of the hottest research topics is efficiently parallelizing the roulette-wheel selection for path construction in ACO. Instead of simply porting from CPU to GPU [22], [78], fine-grained data-parallel approaches were proposed in [79] and [80]. More complicated data structures and memory usage were also introduced for better performance [81], [82]. Based on parallel prefix-sum, some authors proposed to exploit fast shared memory [83] and reduce the use of random numbers [84] to further improve the performance.

Cecilia *et al.* [80] observed the existence of redundant computation and thread divergences in such task-parallel approach used in [22]. A fine-grained data-parallel approach was used to enhance construction performance for solving the travelling salesman problem (TSP). A new method called independent roulette (I-Roulette) was proposed to replicate the classic roulette wheel while improving GPUs parallelism. In this design, Intercity probabilities need to be calculated only once and stored into additional memory to avoid repeated computation. A tiling technique was proposed to scale this method to problems with large numbers of cities [80]. Experiments showed that the quality of solution obtained by I-Roulette ACO was comparable to roulette-wheel-based ACO for solving TSP. More details on data structures and memory usage were discussed in [81] and [82]. Before the work of Cecilia *et al.* [80], Fu *et al.* [79] implemented the same method which the authors called all-in-roulette by using MATLAB Jacket.

Following the work in [80], Dawson and Stewart [83] proposed a novel parallel implementation of roulette-wheel selection algorithm (Double-Spin Roulette, DS-Roulette) to tackle some drawbacks of I-Roulette. DS-Roulette, in effect, is a two-phase prefix-sum-based implementation of roulette-wheel selection by leveraging the fast shared memory. Experiments showed that it greatly reduced the execution time for constructing paths.

Uchida *et al.* [84] described a group of implementation strategies for path construction. Different from [80], only one random number is needed and a straightforward roulette-wheel based on prefix-sum is adopted for city selection. A compressed mechanism was proposed to remove the visited cities before prefix-sum. To avoid performing prefix-sum for every selection (as the case of the former two methods and [83]), stochastic trial was proposed. In this proposal, before path construction starts, the prefix sums for each city are calculated. When selecting city, regular roulette wheel selection is performed until an unvisited city is picked out. Finally, a hybrid method of the three methods was proposed for better performance.

As roulette-wheel selection is widely used for other population-based heuristic algorithms, so the proposed techniques may be used in this areas such as GA and artificial bee colony [85].

5) Pheromone Update: Pheromone update in ACO is also an active research topic. Pheromone update comprises two major tasks: a) pheromone evaporation and b) pheromone deposition. While the parallelization of pheromone is trivial, pheromone deposition is kind of problematic, as different ants may try to depot pheromone onto the same edge at the same time. Both atomic operation-based [84] and atomic-free [80] strategies have been proposed.

The straightforward solution to parallelize pheromone deposition is to prevent race conditions when accessing the pheromone matrix. In this manner, Cecilia *et al.* [80] implemented pheromone update directly on GPU. Another implementation based on atomic operation was proposed by Uchida *et al.* [84]. In this design, the atomic operations happen in shared memory instead of in the global memory as the case of [80].

Cecilia *et al.* [80] proposed a scatter to gather transformations technique to perform pheromone deposition without atomic operations, at the cost of drastically increasing the number of accesses to device memory [$O(n^4)$ in comparison with $O(n^2)$ in the atomic-instruction implementation]. Experimental results in [80] and [82] showed that this implementation is significantly inefficient ($\sim$tenfold slower) than the atomic operation-based implementation.

6) Other Parallelism: For a particular SI algorithm, special consideration is necessary for the GPU-based parallel implementation.

Local search is an optional component for ACO. This mechanism can improve the solution quality greatly at the cost for enormous computation time. As each solution is being improved independently of others, this step is very suitable for task parallel. Tsutsui and Fujimoto [86]–[88] proposed an ACO variant with tabu search and 2-opt search as local search component, respectively. A technique called move-cost adjusted thread assignment was introduced to further accelerate calculating the cost in the process of local search at the cost of extra memory space.

D. All-GPU Parallel Model

Compared to the fast computational speed of GPU, the communication between GPU and CPU is very slow. The overhead of frequent kernel launch is also a potential factor that may harm GPUs overall efficiency. So it may be beneficial to combine multiple kernels into single one thus run a whole program on the GPU only (see Fig. 6). In this case, serial code is deported onto the GPU. Though it is not the priority

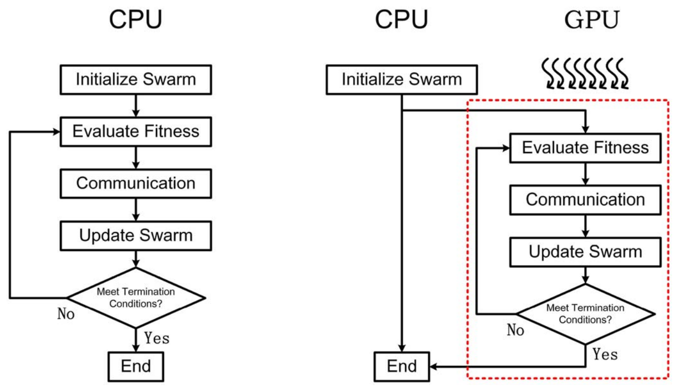


Fig. 6. All GPU parallel model.

of GPU, compared to the communication and kernel launch overhead, the offload will pay off if the serial code is not too computational intensive, thus the overall performance can get improved.

1) Coarse-Grained Strategy: One difficulty that prevents kernel from merging is the data synchronization. Data dependency may exist between different phases. Before a phase starts, the previous phase must complete and the data must be written back correctly. For instance, in PSO with global topology, only until particles have completed updating their private best (position and velocity) and all data are written back to the proper memory, the operation of finding the best particle can start.

However, GPU cannot synchronize between different thread blocks, so we must stop the kernel to explicitly make sure the synchronization just as in the common cases of multiphase parallel model.

To tackle this issue, a direct way is to organize all thread into one single block, i.e., the swarm is mapped onto one thread block, and each particle is mapped onto one thread [89], [90]. A bonus of such a coarse-grained design is that fast shared memory can be used for communication thus better performance can be obtained.

However, as a block must be assigned to one single streaming multiprocessors (SM) and one SM can only support limited threads and blocks. This strategy is only reasonable for small swarm size. So, coarse-grained all-GPU parallel strategy is oftentimes adopted as a component of multiswarm parallel model, which will be discussed below.

2) Fine-Grained Strategy: As aforementioned, if fitness function can be implemented in parallel, fine-grained parallel can be taken, in which case each particle is mapped onto one thread block. Each thread execute some part of function evaluation and the partial results are reduced to give the final value.

As the hardware synchronization is not supported by GPU, two solutions can be used to tackle the issue: a) remove data-dependency and b) utilize software synchronization. An implementation of the former idea is from [91] where the authors proposed PSO without interparticle communication. To the best of our knowledge, there is no published work using software synchronization. But, we notice that persistent thread [92] is a handy tool for this purpose.

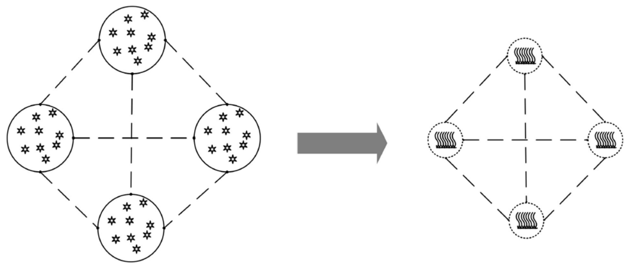


Fig. 7. Multiswarm parallel model.

E. Multiswarm Parallel Model

SIAs can be plagued by a rapid deterioration of their optimization capabilities as the problem scale and dimensionality of the search space increases. The effective approach consists of decomposing the problem in smaller subproblems and then running on them multiple search instances, which corporate with each other by sharing information. Meanwhile, as a computing device with hundreds, even thousands of processing cores, GPU can support enormous number of threads at the same time, thus is able to run thousands of agents simultaneously.

So for high-dimensional, large-scale problems, multiswarm parallel model comes as a natural way not only to reduce the search time but also to improve the quality of the solutions provided. Instead of running one single swarm, in multiswarm parallel model, the swarm is divided into a few subswarms, and each of them evolves separately utilizing different thread or group of threads (see Fig. 7). In literature, multiswarm island model can be called as island model alternatively. Compared to a single swarm with a large population, algorithms using multiple smaller swarms may perform better in terms of both solution quality and running time [93].

Multiswarm model can be divided into two groups: 1) autonomous multiswarm model, where multiple independent copies of the same algorithm execute in parallel and swarms are free to converge toward different suboptima and 2) collaborative multiswarm model, where communication of certain form is allowed among swarms.

1) Autonomous Multiswarm Model: Multiswarm implementations can be memory bounded, thus introduce challenges for GPU platform.

One strategy to address this issue is to run multiple subswarms independently, and choose the best solution of all [94], [95]. This strategy can be helpful for better maintaining the multimodal distribution [96]. Another approach is to divide the original problems into multiple nonoverlapping subproblems and utilize multiple swarms to tackle each subproblems, respectively, [97]–[100].

2) Collaborative Multiswarm Model: Although suffering from the drawbacks of data transfer and synchronization overhead, it may be beneficial for multiple swarms cooperation instead of total independence.

Much attention is focused on efficient immigration strategies. The immigration can be in the form of regrouping of the whole population [101] or more typically, replacing the some individuals (e.g., the worst ones) with the particular individuals (e.g., better ones) from other swarms. The immigrated individuals can maintain their critical information (like velocities for PSO) [102], [103] or discard such information [104], [105]. In some proposals, no individual immigrating happens, instead of the global best information interchanged among swarms [106], [107]. Some proposed that swarms communicate through a master swarm [108], which leads to a more implicit communication mechanism.

The cooperation can also be in the form of space partition. By occupying nonoverlapping space, the whole population can search the solution space more efficiently. The space can be partitioned in a determinant way [26], [73] or stochastically [25].

IV. Some Considerations in Implementation

In this section, we briefly describe several important optimization techniques concerned when implementing SIAs on GPU.

A. Float Numbers: Double Precision or Single Precision

Initially designed for graphics tasks which just need a low precision, GPUs own enormous computing capability for single precision (SP) float operations. The support to double precision (DP) float operations is a relative new thing in recent few years. DP float operations can be 1/3 to 1/16 slower on various particular hardware, and double memory space are needed for storage.

So if SP float operation can satisfy the precision requirement, running algorithms on SP float can fully leverage GPUs' whole computing power. For the same reason, if low precision is acceptable, then the faster native math functions can be used as possible as it can. Some mathematical operators (such as powf, sin) can drastically affect the overall performance of GPUs [109], fast low-precision native implementation (usually several fold faster than the conventional implementation) can improve the performance greatly but the precision loss depends on the specific device [82].

In one word, SP float operation and native functions should be the first consideration issue whenever the precision can satisfy the task at hand.

B. Memory Accesses

The GPU has a hierarchical memory structure as illustrated by Fig. 8. Threads may access data from multiple memory spaces during their execution. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. Texture and constant memories are read-only, and are cached for fast access.

Different memories are quite different with respect to bandwidth (see Table I for the theoretical peak bandwidth). GPU can access host's memory via system bus (Table I lists the peak bandwidth for PCIe 2.0 bus with 16 lanes, which is widely used for today's PC). Global memory are off-chip DRAM which is usually connected to the GPU via GDDR5. Shared memory, in effect, is a block of programmable on-chip L1

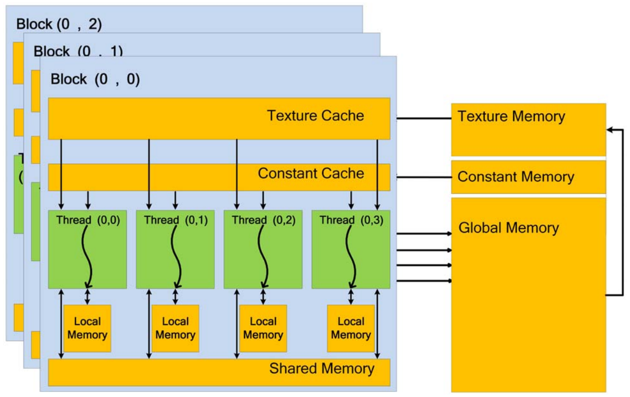


Fig. 8. Memory hierarchy of GPU.

TABLE I
Data Transfer Rates for GPU (NVIDIA GeForce 560 Ti, PCIe 2.0 x16)

	Memory	Global Memory	Shared Memory	Local Memory
Bandwidth	16 GB/s	128.26 GB/s	1024 GB/s	no latency

cache with limited capacity. Local memory is the register, thus can be accessed without latency.

In summary, each memory obeys its specific access pattern, so memory traffic must be carefully tuned to exploit the full bandwidth from all memory controllers.

C. Random Number Generation

In early proposals such as [23], all random numbers were generated on the CPU and then transfer to graphics memory. Random numbers can be efficiently generated directly on the GPU in highly parallel manner [110], and handy library is also available [111], [112]. In the common CPU-based implementations, random numbers are typically generated as needed. In the case of GPUs, it is much more efficient to generate a bunch of random numbers once. For fast access, the random numbers can be stored in read-only memory.

D. Branch Divergence

All GPU threads are organized into multiple groups (warps or wavefronts). Threads in the same group execute the same instruction when selected [52]. Divergence between threads within the same group can cause significant performance bottlenecks for GPU applications. So when designing algorithms for the GPU platform, divergence should be avoided whenever possible.

Irregular loops and conditional instructions are the major sources of thread divergence. Thus, the sequential code, in practice, needs to be restructured delicately to eliminate divergence. Some techniques are proved to be very useful to reduce branch divergence [113]. The iteration delaying trick improves performance by executing loop iterations that take the same branch direction and delaying those that take the other direction until later iterations. The trick called branch distribution reduces the length of divergent code by factoring out structurally similar code from the branch paths.

Divergence can be eliminated by forcing all threads in a warp to take the same control path. The result is an implementation that does not agree with the corresponding sequential version. In particular circumstances where the applications can tolerate errors to some extent, such implementation can still produce acceptable outputs. To safely and efficiently support this strategy, a whole framework and methodology are proposed in [114].

V. Evaluation Criteria

Once an implementation is proposed, a critical question is how well such a proposal performs and what the advantages it has over other implementations. In this section, we discuss the criteria used in literature and give our comments on the rationalities and irrationalities. Based on these observations, we propose our criteria for general fair comparison.

A. Parallel Performance Metrics

Speedup and efficiency are the most common metrics used by the parallel computing community. Speedup can be defined as the ratio of the execution time of parallel implementation (T_p) and the execution time of the sequential one (T_s) (1), and efficiency is a normalized value of the speedup regarding the number of cores (m) executing a parallel algorithm (2)

$$S = \frac{T_s}{T_p} \tag{1}$$

$$E = \frac{S}{m}. \tag{2}$$

In conventional parallel computing, the speedup allows to evaluate how faster a parallel algorithm is compared to a corresponding sequential algorithm while the efficiency gives a platform-independent metric of different parallel implementations. But, in GPU computing, the two metrics have different meanings.

As the execution times are tested on hardware platforms with totally different architectures—the CPU and the GPU, different researchers can use different GPUs and different CPUs, thus making the comparison of different implementations very hard if not impossible. Similarly, the efficiency is not as a useful metric as in CPU parallel analysis.

In some cases, the speedup is calculated on an unfair base. Considerable effort is spent to optimize the code on the GPU while no effort whatsoever is made to optimize the sequential code on the CPU. In such case, inflating speedups by comparing to an unoptimized sequential code, happens too often and has led to unrealistic expectations of the benefits of moving from a CPU to a GPU [115]. Fortunately, the issue of speedup inflation has been noticed and framework for fair comparison has been proposed in [116] and [117].

B. Algorithm Performance Metrics

In conventional framework of CPU-based algorithms, there are two major ways to compare the performance of two algorithms. One way is to compare the accuracy of two algorithms to solve the same problem for a fixed number of function evaluations. Another way is to compare the numbers of function evaluations required by two different algorithms for a given accuracy. An implicit assumption underlying such a method is that, the number of function evaluations roughly reflect the running time of the algorithms.

However, such assumption does not hold when parallelization gets involved and abuse of the two methods leads to some confusions in the study.

In literature, solution quality achieved by GPU-based implementation is always compared with the CPU counterpart with the same population size and evaluation times. If the problem to be solved is very complex, a normal population size can fully exploit the computational power, then this comparison makes sense. However, oftentimes the fitness function is not computationally intensive enough or problem scale is moderate. In such case, significant speedup is obtained only when population size is very large. Such a large population size is usually far from the reasonable one for conventional CPU configuration. For many swarm algorithms, the reasonable population size is relatively moderate. For instance, PSOs population size is usually between 20 to 100 [7].

The conventional iteration-oriented comparison is problematic in the context of GPU computing. In a viewpoint of practice (this is what SIAs are all about), the speedup calculated with this assumption cannot reflect its usage in real-world application. Subsequently, we will present our opinions for a practical criterion for comparison.

C. Proposed Criteria

As the established comparison methods have drawbacks, we propose new criteria to improve the evaluation.

To evaluate the parallel performance, we propose a rectified efficiency to evaluate the parallel performance across different CPU and GPU platforms. To calculate the modified efficiency, we need the theoretical processing power ratio (denoted as R) of the target CPU and GPU. R is defined as

$$R = \frac{P_{\mathrm{gpu}}}{P_{\mathrm{cpu}}} \tag{3}$$

where P_{gpu} and P_{cpu} are the theoretical peak processing power (number of float operations per second) of GPU and CPU, respectively.

Rectified efficiency (RE) can be calculated as follows:

$$\mathrm{RE} = \frac{S}{R} \tag{4}$$

where S is the speedup defined by (1). The larger RE is, the more efficient the parallel implementation is.

Notice that in the scenario of symmetric multicore CPU, $R = (m * P_{\mathrm{cpu}})/P_{\mathrm{cpu}} = m$. so the rectified RE can be viewed as a generalization of efficiency E, thus play a similar role as E for parallel performance evaluation.

We suggest that the rectified efficiency can be used with efficiency to compare the parallel performance. On one hand, the efficiency (multithreaded implementation against single-threaded implementation) functions as the quality of the CPU-based parallel implementation. On the other hand,

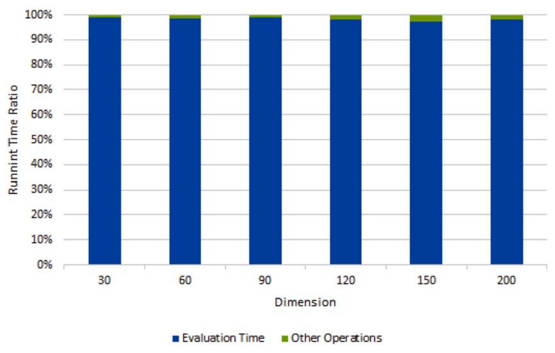


Fig. 9. Breakdown timing.

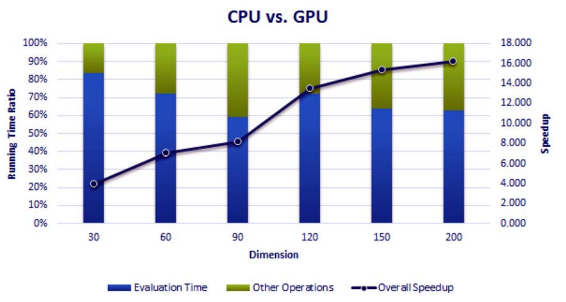


Fig. 10. Overall speedup achieved by GPU-based implementation using naive model.

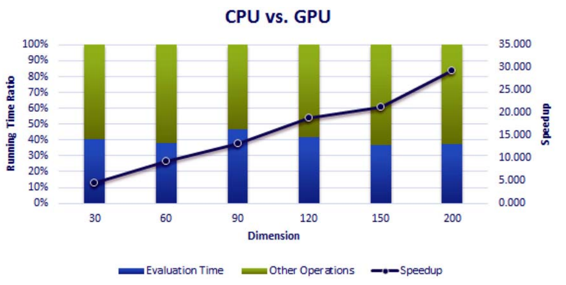


Fig. 11. Overall speedup achieved by GPU-based implementation using multiphase model.

the rectified efficiency (GPU implementation against multi-threaded implementation) monitors the GPU-based parallel implementation.

As for the algorithm performance, it is more reasonable to compare the accuracy under limited time or compare the consumed time before a given accuracy is achieved. The benefit GPU brings is either accuracy improvement (within limited time) or acceleration (with given accuracy). For insightfulness and practical comparison, both CPU and GPU implementations should be fully optimized. Based on that, we can observe, how good the solution is with limited time or how long it will take to reach a particular quality.

VI. Case Study

In this section, the parallel models and performance metrics proposed in this paper are studied further with experiments.

Without a specific note, the experiments are conducted with the following setting: a PC running 64-bit Windows 7 Professional with 8G DDR3 Memory and Intel core I5-2310 (@2.9 GHz 3.1 GHz) and NVIDIA GeForce GTX 970. The program was implemented with C and compiled with visual studio 2013 and CUDA 6.5.

A. Case Study on GPU Implementation

In this case study, we will optimize the Weierstrass function (5) using PSO. Following the optimization methodology and parallel models introduced in Section III, we will see how we can achieve significant speedup at the end.

Weierstrass function is defined as follows:

$$f(\mathbf{x}) = \sum_{i=1}^{D}\sum_{k=0}^{k_{\max}} a^k \cos\left(2\pi b^k(\mathbf{x}_i + 0.5)\right) \tag{5}$$

where $a = 0.5$, $b = 3$, $k_{\max} = 20$, and D denotes the dimension. Weierstrass function is a widely used benchmark function for optimization. It is multimodal and continuous everywhere but differentiable on no point.

In our case, we use standard PSO with ring topology [7] with a fixed size population (100) to optimize Weierstrass function. Fig. 9 illustrates the ratio of function evaluations and other operations. As can be seen, in all setup, the function evaluation takes up around 99% of the running time.

Obviously, the hotspot of this task is about the function evaluation. Thus by following the naive model, we port the evaluation onto the GPU side with a fine-grained implementation. Fig. 10 presents the speedup achieved. Up to 16× speedup is obtained. (Note that the CPU version is parallelized using OpenMP which is about 3.5 times fast than the sequential version.)

Observing Fig. 10, after parallelizing the function evaluation, the ratio of function evaluation takes up only about 60% of the overall running time. The time consumed by other operations cannot be ignored any more. Following the multiphase model, we parallelize the velocity update and position update on the GPU side. The speedup achieved by this implementation is illustrated in Fig. 11. The speedup increase from the naive model's 16× to about 30×. (Note that in our case, by using multiphase model, the running time of function evaluation is reduced further. This improvement is due to the fact that by using the multiphase model, all data are stored on the GPU side, thus the overhead of data transfer between CPU and GPU is eliminated.) By the definition of rectified efficiency, the multiphase implementation achieves an RE value of around 0.7. The result implies that the multiphase implementation is very efficient with the given hardware and task.

In this case study, multiphase model is sufficient for the task. In some cases, a fine-tuned parallelization of the objective function alone may result in a speedup that is sufficient enough. In other cases, say, with more time-consuming task and more powerful devices (e.g., multiple GPUs or GPU clusters), multiswarm model combining with all-GPU model might be used for further exploiting the power of parallelization.

B. Case Study on Criteria

In this section, we will revisit a pioneering work on GPU-based PSO [23] with the criteria introduced in Section V

TABLE II
SPEEDUP AND RECTIFIED EFFICIENCY WITH VARIOUS POPULATION SIZES ($f1$, DIMENSION = 50)

POP	Case 1		Case 2(A)		Case 2(B)	
	S	RE	S	RE	S	RE
400	3.74	0.10	4.37	0.32	1.59	0.12
1200	4.64	0.12	11.12	0.82	4.20	0.31
2000	4.54	0.12	18.43	1.35	6.62	0.49
2800	4.98	0.13	23.79	1.75	8.08	0.59

to verify their rationality in evaluating GPU-based SIAs. It will be clear very soon that with the help of the proposed RE concept, this early implementation was not efficient enough. Based on the observation, we come up with an improved version. Thanks to the new criteria, we can prove that is more efficient than the original one despite of the fact that we used a very different hardware with respect to our task at hand.

1) Case Study on Parallel Performance: Zhou and Tan [23] proposed one of the earliest GPU-based PSO implementation and discussed the speedup with respect to three benchmark functions ($f1 \sim f3$). The speedups (S) measured in this paper are listed in Table II (case 1). The speedups were tested on $f1$ (sphere function) with dimension 50 and various population sizes (rows). We calculate the RE according to the hardware specification (NVIDIA 8600 GT and Intel Core Duo 2).

If only speedup is considered, the GPU implementation seems quite good as considering the used GPU is relatively less powerful than the updated GPUs. However, two factors make this conclusion susceptible and unreliable. First, although CPU of two cores was used in the experiments, only single thread implementation was used for comparison. As aforementioned, this was an unfair comparison, and speedup is overestimated. Second, as the rectified efficiency implies (notice that the RE is also overestimated due to the inflating speedups), the power of GPU was not fully exploited in comparison with CPU. The inefficiency could be caused by at least two obvious factors: a) one critical component of PSO, random number generator, is implemented by the CPU, which is much less efficient than its GPU-based counterparts [110] and b) another factor is that, the CUDA driver then was not as efficient as nowadays.

To address these issues, we improve this implementation by generating random numbers on the GPU directly. Besides, in order to conduct a fair comparison, we implement CPU version with both single thread and multiple threads (using OpenMP) with different hardware (NVIDIA 560 Ti and Intel i5). The results are listed in Table II [case 2(A) for speedups against single-threaded while case 2(B) for multithreaded].

Comparing the REs of cases 1 and 2(A), it is easy to conclude that the new implementation improved the original implementation greatly, while such a conclusion cannot be achieved if only speedup is concerned. According to the RE of case 2(B), the efficiency of the new implementation is still not very plausible, thus more dedicated implementation can be expected.

These analyses hold for other experimental settings, as illustrated by Figs. 12 and 13. As $f2$ and $f3$ are more computationally intensive than $f1$, the GPU achieved better speedup and higher rectified efficiency.

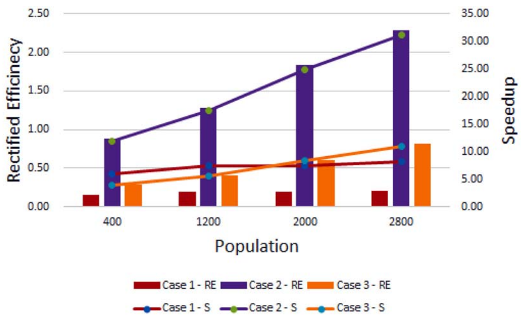


Fig. 12. Speedup and rectified efficiency with various population sizes ($f2$, dimension = 50).

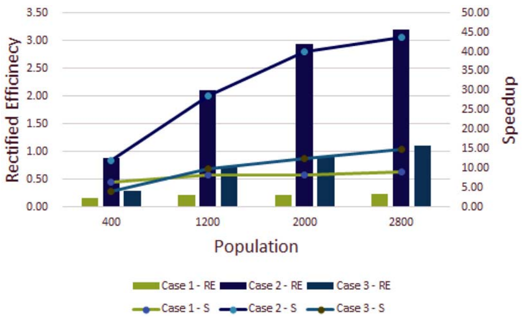


Fig. 13. Speedup and rectified efficiency with various population sizes ($f3$, dimension = 50).

2) Case Study on Algorithm Performance: To show how GPU can significantly improve SIAs' performance, in this case study, we try to utilize PSO to find the minimal of Weierstrass function defined by (5) and Ackley function [see (6)] with $D = 50$

$$f(\mathbf{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D}\mathbf{x}_i^2}\right) - \exp\left(\frac{1}{D}\prod_{i=1}^{D}\cos(2\pi\mathbf{x}_i)\right). \tag{6}$$

Multithreaded and GPU-based PSO implementations in the last case study are reused here, and the testbed is also identical. For the CPU case, 10 000 function evaluations were conducted with different population sizes, for each of which 35 independent trials were run. The GPU-based algorithm ran for roughly the identical time (around 11 s for each population size). Results are illustrated in Fig. 14. It can be seen that GPU-based PSO could obtain better or comparative solutions in comparison with its CPU-based counterpart, even when the population size is relatively small.

Based on the above results, we can make some insightful conclusions below.

a) For CPU-based PSO, it is better to choose population of moderate size (for instance 50 in our case), which agree with the conventional empirical rule.
b) However, this rule of thumb is invalid for the GPU case. In our case, the best solution was achieved when the population size is 200 and 1000 for Ackely function and Wierstrass function, respectively.

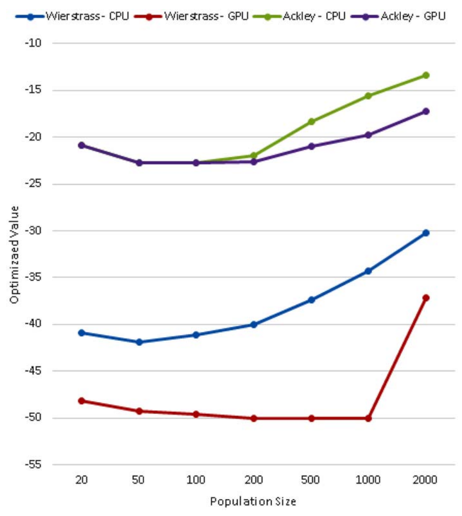


Fig. 14. Solutions obtained under different conditions.

c) The speedup achieved by using GPU can lead to performance improvement, which is very important for GPUs applications in real-world problems.
d) Cares should be taken when pursuing high speedup because high speedup with very large population may deteriorate performance badly, just as our case when population size is 2000.

As seen from the case study above, the new criteria can result in more objective and insightful evaluation for GPU-based implementation, thus help us deploying GPU-based SIAs in practical applications. For more details, please visit the FWA forum at our CIL@PKU at http://www.cil.pku.edu.cn/research/fwa/index.html.

VII. Conclusion

In this paper, an extensive literature review of GPU-based SIAs was presented. All proposals have been grouped into four categories in a viewpoint of optimization. Several important concerns in implementation were described in detail along with the discussion of metrics evaluating parallelization and algorithm performance.

As we have come into an era of multi- and many-core era, single thread is not a reasonable assumption any more. Parallel implementation will dominate the design and implementation of SIAs both on CPU and GPU. Little interest and attention will the sequential version have both for academic research and industrial applications.

Although performance measures are key in evaluating and improving GPU-based parallel algorithms, good performance criteria are yet to be developed. A rectified efficiency was proposed in this paper to measure parallel performance. As for the algorithm performance, it is more reasonable and practical to compare solution quality under limited time or compare the consumed time given accuracy.

Multiobjective optimization (MOO) is always a hot topic in SI community, however, only few implementations (several proposals can be found in [36], [75], [118], and [126]) leveraged GPUs computational powers. Because MOO problems are, usually, much more computationally intensive than single-objective ones, thus accelerating these MOO problems will be very useful and beneficial for algorithms and applications.

As GPUs have been used to solve large-scale problems successfully, multi-GPU and GPU cluster will become popular for implementations of SIAs in future. Several preliminary works have been reported [42], [119]. Fortunately, the hardware infrastructure is mature and software is ready for use [120], [121].

Although CUDA is the dominate platform for GPU-based implementation of SIAs, it suffers from the drawback of closed environment as only NVIDIAs GPUs were supported. Since diverse hardware platforms are at hand for accelerating SIAs, from embedded systems [122], [123], [125] to super computers, a more universal solution will be applaudable. OpenCL could be a good option to address this need and a competitive alternative for CUDA. OpenCL can be as efficient and productive as CUDA is [124]. Besides, supported by multiple vendors, OpenCL also enjoys the advantage of portability. Without modification, a program written in OpenCL can run on all these hardware. In future, more researches and applications would be based on OpenCL platform.

Finally, it would be pointed out that there is no discussion on energy consumption in literature, to the best of our knowledge. Although GPU was reported more energy-efficient compared to CPU for supercomputers, it is still an open question, i.e., how GPU performs in the scenario of low power system (e.g., embedded and mobile devices).

References

[1] R. C. Eberhart, Y. Shi, and J. Kennedy, *Swarm Intelligence*. San Francisco, CA, USA: Morgan Kaufmann, 2001.
[2] Y. Tan, *An Introduction to Fireworks Algorithm*. Beijing, China: Science Press, 2015.
[3] Y. Tan, *Fireworks Algorithm: A Novel Swarm Intelligence Method*. Berlin, Germany: Springer, 2015.
[4] A. P. Engelbrecht, *Fundamentals of Computational Swarm Intelligence*. Hoboken, NJ, USA: Wiley, 2005.
[5] X.-S. Yang, "Swarm intelligence based algorithms: A critical analysis," *Evol. Intell.*, vol. 7, no. 1, pp. 17–28, Apr. 2014.
[6] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proc. IEEE Int. Conf. Neural Netw.*, vol. 4. Perth, WA, Australia, Nov./Dec. 1995, pp. 1942–1948.
[7] D. Bratton and J. Kennedy, "Defining a standard for particle swarm optimization," in *Proc. IEEE Swarm Intell. Symp. (SIS)*, Honolulu, HI, USA, Apr. 2007, pp. 120–127.
[8] M. Dorigo and L. Gambardella, "Ant colony system: A cooperative learning approach to the traveling salesman problem," *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 53–66, Apr. 1997.
[9] M. Dorigo and C. Blum, "Ant colony optimization theory: A survey," *Theor. Comput. Sci.*, vol. 344, nos. 2–3, pp. 243–278, Nov. 2005.
[10] Y. Tan and Y. Zhu, "Fireworks algorithm for optimization," in *Advances in Swarm Intelligence* (LNCS 6145). Berlin, Germany: Springer, 2010, pp. 355–364.
[11] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. W. Brodersen, "Optimizing power using transformations," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 14, no. 1, pp. 12–31, Jan. 1995.
[12] P. Ross, "Why CPU frequency stalled," *IEEE Spectr.*, vol. 45, no. 4, p. 72, Apr. 2008.
[13] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*. San Francisco, CA, USA: Morgan Kaufamnn, Aug. 2011.

[14] R. Buchty, V. Heuveline, W. Karl, and J.-P. Weiss, “A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators,” *Concur. Comput. Pract. Exp.*, vol. 24, no. 7, pp. 663–675, May 2012.

[15] R. Weber, A. Gothandaraman, R. Hinde, and G. Peterson, “Comparing hardware accelerators in scientific applications: A case study,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 58–68, Jan. 2011.

[16] D. B. Kirk and W. mei Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Beijing, China: Tsinghua Univ. Press, 2010.

[17] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming Guide*. Upper Saddle River, NJ, USA: Addison-Wesley Professional, 2011.

[18] J. D. Owens *et al.*, “A survey of general-purpose computation on graphics hardware,” *Comput. Graph. Forum*, vol. 26, no. 1, pp. 80–113, Feb. 2007.

[19] AMD Inc. (2015). *Application Showcase*. [Online]. Available: http://developer.amd.com/community/application-showcase/

[20] NVIDIDA Corp. (2015). *CUDA in Action—Research & Apps*. [Online]. Available: https://developer.nvidia.com/cuda-action-research-apps

[21] J. Li, D. Wan, Z. Chi, and X. Hu, “A parallel particle swarm intelligence algorithm based on fine-grained model with GPU-accelerating,” *J. Harbin Inst. Technol.*, vol. 38, no. 12, pp. 2162–2166, Dec. 2006.

[22] A. Catala, J. Jaen, and J. Mocholi, “Strategies for accelerating ant colony optimization algorithms on graphical processing units,” in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Singapore, 2007, pp. 492–500.

[23] Y. Zhou and Y. Tan, “GPU-based parallel particle swarm optimization,” in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Trondheim, Norway, May 2009, pp. 1493–1500.

[24] W. Zhu and J. Curry, “Parallel ant colony for nonlinear function optimization with graphics hardware acceleration,” in *Proc. IEEE Int. Conf. Syst. Man Cybern. (SMC)*, San Antonio, TX, USA, 2009, pp. 1803–1808.

[25] K. Ding, S. Zheng, and Y. Tan, “A GPU-based parallel fireworks algorithm for optimization,” in *Proc. 15th Annu. Conf. Genet. Evol. Comput. Conf. (GECCO)*, Amsterdam, The Netherlands, 2013, pp. 9–16.

[26] G.-H. Luo, S.-K. Huang, Y.-S. Chang, and S.-M. Yuan, “A parallel Bees Algorithm implementation on GPU,” *J. Syst. Archit.*, vol. 60, no. 3, pp. 271–279, Mar. 2014.

[27] R. Jovanovic and M. Tuba, “Parallelization of the Cuckoo search using CUDA architecture,” in *Proc. 7th Int. Conf. Appl. Math. Simulat. Model. (ASM)*, Cambridge, MA, USA, Mar. 2013, pp. 137–142.

[28] J. Reguera-Salgado and J. M.-Herrero, “High performance GCP-based particle swarm optimization of orthorectification of airborne pushbroom imagery,” in *Proc. IEEE Int. Geosci. Remote Sens. Symp. (IGARSS)*, Munich, Germany, 2012, pp. 4086–4089.

[29] A. Kristiadi, P. Pranowo, and P. Mudjihartono, “Parallel particle swarm optimization for image segmentation,” in *Proc. 2nd Int. Conf. Digit. Enterp. Inf. Syst. (DEIS)*, Kuala Lumpur, Malaysia, Mar. 2013, pp. 129–135.

[30] L. Dawson and I. A. Stewart, “Accelerating ant colony optimization-based edge detection on the GPU using CUDA,” in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Beijing, China, 2014, pp. 1736–1743.

[31] B. Rymut and B. Kwolek, “Real-time multiview human body tracking using GPU-accelerated PSO,” in *Parallel Processing and Applied Mathematics* (LNCS 8384). Berlin, Germany: Springer, 2014, pp. 458–468.

[32] L. Mussi, S. Ivekovic, and S. Cagnoni, “Markerless articulated human body tracking from multi-view video with GPU-PSO,” in *Evolvable Systems: From Biology to Hardware* (LNCS 6274). Berlin, Germany: Springer, 2010, pp. 97–108.

[33] Z. Zhang and H. S. Seah, “CUDA acceleration of 3D dynamic scene reconstruction and 3D motion estimation for motion capture,” in *Proc. IEEE 18th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Singapore, 2012, pp. 284–291.

[34] R. Ugolotti *et al.*, “Particle swarm optimization and differential evolution for model-based object detection,” *Appl. Soft Comput.*, vol. 13, no. 6, pp. 3092–3105, Jun. 2013.

[35] J. Platos, V. Snasel, T. Jezowicz, P. Kromer, and A. Abraham, “A PSO-based document classification algorithm accelerated by the CUDA platform,” in *Proc. IEEE Int. Conf. Syst. Man Cybern. (SMC)*, Seoul, Korea, 2012, pp. 1936–1941.

[36] A. Cano, J. L. Olmo, and S. Ventura, “Parallel multi-objective ant programming for classification using GPUs,” *J. Parallel Distrib. Comput.*, vol. 73, no. 6, pp. 713–728, Jun. 2013.

[37] C. Zhang, H.-C. Mei, and H. Yang. (2014). *A Parallel way to Select the Parameters of SVM Based on the Ant Optimization Algorithm*. [Online]. Available: http://arxiv.org/abs/1405.4589

[38] R. M. Weiss, “GPU-accelerated ant colony optimization,” in *GPU Computing Gems*. Waltham, MA, USA: Morgan Kaufamnn, 2011, pp. 325–340.

[39] R. M. Weiss, “Accelerating swarm intelligence algorithms with GPU-computing,” in *GPU Solutions to Multi-scale Problems in Science and Engineering* (Lecture Notes in Earth System Sciences). Berlin, Germany: Springer, 2013, pp. 503–515.

[40] D. Datta, S. Mehta, Shalivahan, and R. Srivastava, “CUDA based particle swarm optimization for geophysical inversion,” in *Proc. 1st Int. Conf. Recent Adv. Inf. Technol. (RAIT)*, Dhanbad, India, 2012, pp. 416–420.

[41] R.-B. Chen, D.-N. Hsieh, Y. Hung, and W. Wang, “Optimizing latin hypercube designs by particle swarm,” *Statist. Comput.*, vol. 23, no. 5, pp. 663–676, Sep. 2013.

[42] O. Kilic, E. El-Araby, Q. Nguyen, and V. Dang, “Bio-inspired optimization for electromagnetic structure design using full-wave techniques on GPUs,” *Int. J. Numer. Model. Electron. Netw. Devices Fields*, vol. 26, no. 6, pp. 649–669, Nov./Dec. 2013.

[43] S. Papadakis and A. G. Bakrtzis, “A GPU accelerated PSO with application to economic dispatch problem,” in *Proc. 16th Int. Conf. Int. Syst. Applicat. Power Syst. (ISAP)*, Hersonissos, Greece, 2011, pp. 1–6.

[44] B. Sharma, R. K. Thulasiram, and P. Thulasiraman, “Portfolio management using particle swarm optimization on GPU,” in *Proc. IEEE 10th Int. Symp. Parallel Distrib. Process. Applicat. (ISPA)*, Leganés, Spain, 2012, pp. 103–110.

[45] B. Sharma, R. K. Thulasiram, and P. Thulasiraman, “Normalized particle swarm optimization for complex chooser option pricing on graphics processing unit,” *J. Supercomput.*, vol. 66, no. 1, pp. 170–192, Oct. 2013.

[46] W. Bura and M. Boryczka, “The parallel ant vehicle navigation system with CUDA technology,” in *Computational Collective Intelligence. Technologies and Applications* (LNCS 6923). Berlin, Germany: Springer, 2011, pp. 505–514.

[47] Y. S. Nashed, R. Ugolotti, P. Mesejo, and S. Cagnoni, “libCudaOptimize: An open source library of GPU-based metaheuristics,” in *Proc. 14th Int. Conf. Genet. Evol. Comput. Conf. Companion (GECCO)*, Philadelphia, PA, USA, 2012, pp. 117–124.

[48] E. G. Cano and K. Rodríguez, “A parallel PSO algorithm for a watermarking application on a GPU,” *Computación y Sistemas*, vol. 17, no. 3, pp. 381–390, Sep. 2013.

[49] A. Ouyang *et al.*, “Parallel hybrid PSO with CUDA for lD heat conduction equation,” *Comput. Fluids*, vol. 110, pp. 198–210, Mar. 2015.

[50] A. Coates *et al.*, “Deep learning with COTS HPC systems,” in *Proc. 30th Int. Conf. Mach. Learn. (ICML)*, Atlanta, GA, USA, May 2013, pp. 1337–1345.

[51] K. Ballou and N. M. Mousa, “Opencuda+mpi,” Stud. Res. Initiat., Montreal, QC, Canada, Tech. Rep. 14, Aug. 2013.

[52] NVIDIA Corp. (Mar. 2015). *CUDA C Programming Guide v7.0* [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[53] Khronos OpenCL Working Group. (Nov. 2011). *The OpenCL 1.2 Specification*. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[54] D. Hallmans, K. Sandstrom, M. Lindgren, and T. Nolte, “GPGPU for industrial control systems,” in *Proc. IEEE 18th Conf. Emerg. Technol. Factory Autom. (ETFA)*, Cagliari, Italy, 2013, pp. 1–4.

[55] A. Maghazeh, U. D. Bordoloi, P. Eles, and Z. Peng, “General purpose computing on low-power embedded GPUs : Has it come of age?” in *Proc. Int. Conf. Embedded Comput. Syst. Archit. Model. Simulat. (SAMOS XIII)*, Samos, Greece, Jul. 2013, pp. 1–10 and 15–18.

[56] E. Aragon, J. M. Jimenez, A. Maghazeh, J. Rasmusson, and U. D. Bordoloi, “Pattern matching in OpenCL: GPU vs CPU energy consumption on two mobile chipsets,” in *Proc. 2nd Int. Workshop OpenCL (IWOCL)*, Bristol, U.K., May 2014, pp. 1–7.

[57] R. Farber, *CUDA Application Design and Development*, 1st ed. Waltham, MA, USA: Morgan Kaufmann, Nov. 2011.

[58] P. Krömer, J. Platoš, and V. Snášel, “A brief survey of advances in particle swarm optimization on graphic processing units,” in *Proc. World Congr. Nature Biol. Inspir. Comput. (NaBIC)*, Fargo, ND, USA, 2013, pp. 182–188.

[59] D. A. Augusto, J. S. Angelo, and H. J. C. Barbosa, “Strategies for parallel ant colony optimization on graphics processing units,” in *Ant Colony Optimization—Techniques and Applications*. Rijeka, Croatia: InTech Open Access, 2013.

[60] E. Alba, G. Luque, and S. Nesmachnow, "Parallel metaheuristics: Recent advances and new trends," *Int. Trans. Oper. Res.*, vol. 20, no. 1, pp. 1–48, Jan. 2013.

[61] P. Krömer, J. Platoš, and V. Snášel, "Nature-inspired meta-heuristics on modern GPUs: State of the art and brief survey of selected algorithms," *Int. J. Parallel Program.*, vol. 42, no. 5, pp. 681–709, Oct. 2014.

[62] F. Valdez, P. Melin, and O. Castillo, "Bio-inspired optimization methods on graphic processing unit for minimization of complex mathematical functions," in *Recent Advances on Hybrid Intelligent Systems* (Studies in computational intelligence), vol. 451, O. Castillo, P. Melin, and J. Kacprzyk, Eds. Berlin, Germany: Springer, 2013, pp. 313–322.

[63] A. Majd and G. Sahebi, "A survey on parallel evolutionary computing and introduce four general frameworks to parallelize all EC algorithms and create new operation for migration," *J. Inf. Comput. Sci.*, vol. 9, no. 2, pp. 97–105, 2014.

[64] NVIDIA Corp. (Mar. 2015). *CUDA C Best Practices Guide v7.0*. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

[65] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proc. Spring Joint Comput. Conf. (AFIPS)*, Atlantic, NJ, USA, Apr. 1967, pp. 483–485.

[66] V. Kalivarapu and E. Winer, "Implementation of digital pheromones in PSO accelerated by commodity graphics hardware," in *Proc. 12th AIAA/ISSMO Multidiscipl. Anal. Optim. Conf.*, Victoria, BC, Canada, Sep. 2008.

[67] M. Cárdenas-Montes, M. A. Vega-Rodríguez, J. J. Rodríguez-Vázquez, and A. Gómez Iglesias, "Accelerating particle swarm algorithm with GPGPU," in *Proc. 19th Euromicro Int. Conf. Parallel Distrib. Netw.-Based Process. (PDP)*, Ayia Napa, Cyprus, 2011, pp. 560–564.

[68] H.-T. Hsieh and C.-H. Chu, "Particle swarm optimisation (PSO)-based tool path planning for 5-axis flank milling accelerated by graphics processing unit (GPU)," *Int. J. Comput. Integr. Manuf.*, vol. 24, no. 7, pp. 676–687, 2011.

[69] I. Blecic, A. Cecchini, and G. A. Trunfio, "Fast and accurate optimization of a GPU-accelerated CA urban model through cooperative coevolutionary particle swarms," *Procedia Comput. Sci.*, vol. 29, pp. 1631–1643, Dec. 2014.

[70] K. Ding and Y. Tan, "CuROB: A GPU-based test suit for real-parameter optimization," in *Advances in Swarm Intelligence* (LNCS 8795). Berlin, Germany: Springer, 2014, pp. 66–78.

[71] Y. Shi, "Reevaluating Amdahl's law and Gustafson's law," Dept. Comput. Sci., Temple Univ., Philadelphia, PA, USA, Tech. Rep. MS 38-24, Oct. 1996.

[72] A. J. C. C. Lins, C. J. A. Bastos-Filho, D. N. O. Nascimento, M. A. C. Oliveira, Jr, and F. B. de Lima-Neto, "Analysis of the performance of the fish school search algorithm running in graphic processing units," in *Theory and New Applications of Swarm Intelligence*. Shanghai, China: InTech, Mar. 2012, pp. 17–32.

[73] A. V. Husselmann and K. A. Hawick, "Parallel parametric optimisation with firefly algorithms on graphical processing units," in *Proc. Int. Conf. Genet. Evol. Methods (GEM)*, Las Vegas, NV, USA, Jul. 2012, pp. 77–83.

[74] Y. Hu, B. Yu, J. Ma, and T. Chen, "Parallel fish swarm algorithm based on GPU-acceleration," in *Proc. 3rd Int. Workshop Intell. Syst. Applicat. (ISA)*, Wuhan, China, 2011, pp. 1–4.

[75] J. Arun, M. Mishra, and S. Subramaniam, "Parallel implementation of MOPSO on GPU using OpenCL and CUDA," in *Proc. 18th Int. Conf. High Perform. Comput. (HiPC)*, Bengaluru, India, 2011, pp. 1–10.

[76] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUS," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, Rome, Italy, 2009, pp. 1–10.

[77] D. Merrill and A. Grimshaw, "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing," *Parallel Process. Lett.*, vol. 21, no. 2, pp. 245–272, 2011. [Online]. Available: http://www.worldscinet.com/ppl/21/2102/S0129626411000187.html

[78] J. Wang, J. Dong, and C. Zhang, "Implementation of ant colony algorithm based on GPU," in *Proc. 6th Int. Conf. Comput. Graphics Imag. Vis.*, Tianjin, China, 2009, pp. 50–53.

[79] J. Fu, L. Lei, and G. Zhou, "A parallel ant colony optimization algorithm with GPU-acceleration based on all-in-roulette selection," in *Proc. 3rd Int. Workshop Adv. Comput. Intell. (IWACI)*, Suzhou, China, 2010, pp. 260–264.

[80] J. M. Cecilia, J. M. Garcia, M. Ujaldón, A. Nisbet, and M. Amos, "Parallelization strategies for ant colony optimisation on GPUS," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops PhD Forum (IPDPSW)*, Anchorage, AK, USA, 2011, pp. 339–346.

[81] J. M. Cecilia, J. M. Garcia, A. Nisbet, M. Amos, and M. Ujaldón, "Enhancing data parallelism for ant colony optimization on GPUS," *J. Parallel Distrib. Comput.*, vol. 73, no. 1, pp. 42–51, 2013.

[82] J. M. Cecilia, A. Nisbet, M. Amos, J. M. García, and M. Ujaldón, "Enhancing GPU parallelism in nature-inspired algorithms," *J. Supercomput.*, vol. 63, no. 3, pp. 773–789, 2013.

[83] L. Dawson and I. A. Stewart, "Improving ant colony optimization performance on the GPU using CUDA," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Cancún, Mexico, 2013, pp. 1901–1908.

[84] A. Uchida, Y. Ito, and K. Nakano, "An efficient GPU implementation of ant colony optimization for the traveling salesman problem," in *Proc. 3rd Int. Conf. Netw. Comput. (ICNC)*, Okinawa, Japan, 2012, pp. 94–102.

[85] H. Narasimhan, "Parallel artificial bee colony (PABC) algorithm," in *Proc. World Congr. Nat. Biol. Inspir. Comput. (NaBIC)*, Coimbatore, India, 2009, pp. 306–311.

[86] S. Tsutsui and N. Fujimoto, "ACO with tabu search on a GPU for solving QAPs using move-cost adjusted thread assignment," in *Proc. ACM 13th Annu. Conf. Genet. Evol. Comput. (GECCO)*, Dublin, Ireland, 2011, pp. 1547–1554.

[87] S. Tsutsui and N. Fujimoto, "ACO with tabu search on GPUS for fast solution of the QAP," in *Massively Parallel Evolutionary Computation on GPGPUs* (Natural Computing Series), S. Tsutsui and P. Collet, Eds. Berlin, Germany: Springer, 2013, pp. 179–202.

[88] S. Tsutsui and N. Fujimoto, "Fast QAP solving by ACO with 2-opt local search on a GPU," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, New Orleans, LA, USA, 2011, pp. 812–819.

[89] L. Mussi, F. Daolio, and S. Cagnoni, "Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture," *Inf. Sci.*, vol. 181, no. 20, pp. 4642–4657, 2011.

[90] R. Calazan, N. Nedjah, and L. M. Mourelle, "Three alternatives for parallel GPU-based implementations of high performance particle swarm optimization," in *Advances in Computational Intelligence* (LNCS 7902). Berlin, Germany: Springer, 2013, pp. 241–252.

[91] L. Mussi, Y. S. Nashed, and S. Cagnoni, "GPU-based asynchronous particle swarm optimization," in *Proc. 13th Annu. Conf. Genet. Evol. Comput. (GECCO)*, Dublin, Ireland, 2011, pp. 1555–1562.

[92] K. Gupta, J. Stuart, and J. Owens, "A study of persistent threads style GPU programming for GPGPU workloads," in *Proc. Innov. Parallel Comput. (InPar)*, San Jose, CA, USA, 2012, pp. 1–14.

[93] G. A. Laguna-Sánchez, M. Olguín-Carbajal, N. Cruz-Cortés, R. Barrón-Fernández, and J. A. Álvarez-Cedillo, "Comparative study of parallel variants for a particle swarm optimization," *J. Appl. Res. Technol.*, vol. 7, no. 3, pp. 292–309, Dec. 2009.

[94] R. Calazan, N. Nedjah, and L. M. Mourelle, "Swarm grid: A proposal for high performance of parallel particle swarm optimization using GPGPU," in *Computational Science and Its Applications* (LNCS 7333). Berlin, Germany: Springer, 2012, pp. 148–160.

[95] M. Rabinovich *et al.*, "Particle swarm optimization on a GPU," in *Proc. IEEE Int. Conf. Elect./Inf. Technol. (EIT)*, Indianapolis, IN, USA, 2012, pp. 1–6.

[96] B. Rymut, B. Kwolek, and T. Krzeszowski, "GPU-accelerated human motion tracking using particle filter combined with PSO," in *Advanced Concepts for Intelligent Vision Systems* (LNCS 8192). Cham, Switzerland: Springer, 2013, pp. 426–437.

[97] L. Mussi, S. Cagnoni, and F. Daolio, "GPU-based road sign detection using particle swarm optimization," in *Proc. 9th Int. Conf. Intell. Syst. Design Appl. (ISDA)*, Pisa, Italy, Nov./Dec. 2009, pp. 152–157.

[98] L. Mussi *et al.*, "GPU implementation of a road sign detector based on particle swarm optimization," *Evol. Intell.*, vol. 3, nos. 3–4, pp. 155–169, 2010.

[99] V. Roberge and M. Tarbouchi, "Efficient parallel particle swarm optimizers on GPU for real-time harmonic minimization in multilevel inverters," in *Proc. 38th Annu. Conf. IEEE Ind. Electr. Soc. (IECON)*, Montreal, QC, Canada, Oct. 2012, pp. 2275–2282.

[100] V. Roberge and M. Tarbouchi, "Parallel particle swarm optimization on graphical processing unit for pose estimation," *WSEAS Trans. Comput.*, vol. 11, no. 6, pp. 170–179, Jun. 2012.

[101] J. Zhao, W. Wang, W. Pedrycz, and X. Tian, "Online parameter optimization-based prediction for converter gas system by parallel strategies," *IEEE Trans. Control Syst. Technol.*, vol. 20, no. 3, pp. 835–845, May 2012.

[102] W. Franz, P. Thulasiraman, and R. K. Thulasiram, "Memory efficient multi-swarm PSO algorithm in OpenCL on an APU," in *Algorithms and Architectures for Parallel Processing* (LNCS 8285). Cham, Switzerland: Springer, 2013, pp. 236–246.

[103] W. Franz, P. Thulasiraman, and R. K. Thulasiram, "Optimization of an OpenCL-based multi-swarm PSO algorithm on an APU," in *Parallel Processing and Applied Mathematics* (LNCS 8385). Berlin, Germany: Springer, 2014, pp. 140–150.

[104] S. Solomon, P. Thulasiraman, and R. K. Thulasiram, "Collaborative multi-swarm PSO for task matching using graphics processing units," in *Proc. 13th Annu. Conf. Genet. Evol. Comput. (GECCO)*, Dublin, Ireland, 2011, pp. 1563–1570.

[105] S. Solomon, P. Thulasiraman, and R. K. Thulasiram, "Scheduling using multiple swarm particle optimization with memetic features on graphics processing units," in *Massively Parallel Evolutionary Computation on GPGPUs* (Natural Computing Series). Berlin, Germany: Springer, 2013, pp. 149–178.

[106] M. Nobile, D. Besozzi, P. Cazzaniga, G. Mauri, and D. Pescini, "A GPU-based multi-swarm PSO method for parameter estimation in stochastic biological systems exploiting discrete-time target series," in *Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics* (LNCS 7246). Berlin, Germany: Springer, 2012, pp. 74–85.

[107] M. S. Nobile, D. Besozzi, P. Cazzaniga, G. Mauri, and D. Pescini, "Estimating reaction constants in stochastic biological systems with a multi-swarm PSO running on GPUS," in *Proc. 14th Int. Conf. Genet. Evol. Comput. Conf. Comp. (GECCO Comp.)*. Philadelphia, PA, USA, 2012, pp. 1421–1422.

[108] D. L. Souza, O. Teixeira, D. C. Monteiro, and R. C. L. Oliveira, "A new cooperative evolutionary multi-swarm optimizer algorithm based on CUDA architecture applied to engineering optimization," in *Combinations of Intelligent Methods and Applications* (Smart Innovation, Systems and Technologies), vol. 23. Berlin, Germany: Springer, 2013, pp. 95–115.

[109] D. Demirovic, A. Serifovic-Trbalic, and P. C. Cattin, "Evaluation of OpenCL native math functions for image processing algorithms," in *Proc. 24th Int. Symp. Inf. Commun. Autom. Technol. (ICAT)*, Sarajevo, Bosnia and Herzegovina, 2013, pp. 1–5.

[110] K. Ding and Y. Tan, "Comparison of random number generators in particle swarm optimization algorithm," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Beijing, China, 2014, pp. 2664–2671.

[111] W. B. Langdon, "A fast high quality pseudo random number generator for nVidia CUDA," in *Proc. 11th Annu. Conf. Compan. Genet. Evol. Comput. Conf. (GECCO)*, Montreal, QC, Canada, 2009, pp. 2511–2514.

[112] NVIDIA Corp. (Mar. 2015). *CURAND Library Programming Guide v7.0*. [Online]. Available: http://docs.nvidia.com/cuda/curand/

[113] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in GPU programs," in *Proc. 4th Workshop Gen. Purpose Process. Graphics Process. Units (GPGPU)*, Newport Beach, CA, USA, 2011, pp. 1–8.

[114] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications," *IEEE Trans. Multimedia*, vol. 15, no. 2, pp. 279–290, Feb. 2013.

[115] V. W. Lee *et al.*, "Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 451–460, Jun. 2010.

[116] S. Cagnoni, A. Bacchini, and L. Mussi, "OpenCL implementation of particle swarm optimization: A comparison between multi-core CPU and GPU performances," in *Applications of Evolutionary Computation* (LNCS 7248). Berlin, Germany: Springer, 2012, pp. 406–415.

[117] V. Roberge and M. Tarbouchi, "Comparison of particle swarm optimization for graphical processing units and multicore processors," *Int. J. Comput. Intell. Appl.*, vol. 12, no. 1, pp. 1–20, 2013.

[118] Y. Zhou and Y. Tan, "GPU-based parallel multi-objective particle swarm optimization," *Int. J. Artif. Intell.*, vol. 7, no. A11, pp. 125–141, Oct. 2011.

[119] S. Tsutsui, "ACO on multiple GPUS with CUDA for faster solution of QAPs," in *Parallel Problem Solving From Nature—PPSN XII* (LNCS 7492), C. Coello *et al.*, Eds. Berlin, Germany: Springer, 2012, pp. 174–184.

[120] G. Shainer *et al.*, "The development of Mellanox/NVIDIA GPUDirect over InfiniBand—A new model for GPU to GPU communications," *Comput. Sci. Res. Dev.*, vol. 26, nos. 3–4, pp. 267–273, 2011.

[121] H. Wang *et al.*, "MVAPICH2-GPU: Optimized GPU to GPU communication for InfiniBand clusters," *Comput. Sci. Res. Dev.*, vol. 26, nos. 3–4, pp. 257–266, 2011.

[122] D. Muñoz, C. Llanos, L. D. S. Coelho, and M. Ayala-Rincon, "Comparison between two FPGA implementations of the particle swarm optimization algorithm for high-performance embedded applications," in *Proc. IEEE 5th Int. Conf. Bio-Inspir. Comput. Theor. Appl. (BIC-TA)*, Changsha, China, 2010, pp. 1637–1645.

[123] D. Muñoz, C. Llanos, L. D. S. Coelho, and M. Ayala-Rincon, "Hardware particle swarm optimization based on the attractive-repulsive scheme for embedded applications," in *Proc. Int. Conf. Reconfigur. Comput. FPGAs (ReConFig)*, Quintana Roo, Mexico, 2010, pp. 55–60.

[124] P. Du *et al.*, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," *Parallel Comput.*, vol. 38, no. 8, pp. 391–407, 2012.

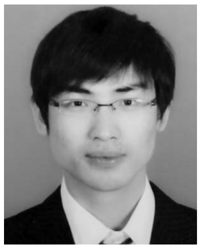
[125] Y. Chen, W. Peng, and M. Jian, "Particle swarm optimization with recombination and dynamic linkage discovery," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 37, no. 6, pp. 1460–1470, Dec. 2007.

[126] D. Liu, K. C. Tan, C. K. Goh, and W. K. Ho, "A multiobjective memetic algorithm based on particle swarm optimization," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 37, no. 1, pp. 42–50, Feb. 2007.

Ying Tan (M'98–SM'02) received the bachelor's, master's, and Ph.D. degrees from Southeast University, Nanjing, China, in 1985, 1987, and 1997, respectively.

He is a Full Professor and the Ph.D. Advisor with the School of Electronics Engineering and Computer Science, and the Director of Computational Intelligence Laboratory, Peking University, Beijing, China. He is the inventor of Fireworks Algorithm. His current research interests include computational intelligence, swarm intelligence, machine learning and data mining, and their applications to information security.

Dr. Tan was a recipient of the Second-Class Natural Science Award of China in 2009. He serves as an Editor-in-Chief of the *International Journal of Computational Intelligence and Pattern Recognition* and an Associate Editor of the IEEE TRANSACTION ON CYBERNETICS, the IEEE TRANSACTION ON NEURAL NETWORKS AND LEARNING SYSTEMS, and the *International Journal of Swarm Intelligence Research* (IJSIR). He also served as an Editor of Springer's Lecture Notes on Computer Science for over 16 volumes and the Guest Editor of several referred journals, including *Information Science*, *Softcomputing*, *Neurocomputing*, *Natural Computing*, the IEEE/ACM TRANSACTIONS ON BIOINFORMATICS AND COMPUTATIONAL BIOLOGY, International Journal of AI, and IJSIR. He is the Foundering General Chair of the series International Conference on Swarm Intelligence from 2010 to 2015, the Joint General Chair of the First and Second BRICS Congress on Computational Intelligence (CCI) in 2013 and 2015, and the Program Committee Co-Chair of the IEEE World CCI 2014.



Ke Ding received the bachelor's degree in computer science from the Chinese University of Agriculture, Beijing, China, in 2010.

He is currently pursuing his Ph.D. degree in computer science from the Key Laboratory of Machine Perception and Intelligence, and the School of Electronics Engineering and Computer Science, Peking University, Beijing, China.

His current research interests include swarm intelligence, graphical processing unit computing, and machine learning.