Hindawi Publishing Corporation
Advances in Meteorology
Volume 2016, Article ID 8483728, 10 pages
http://dx.doi.org/10.1155/2016/8483728

Research Article

Accelerating the SCE-UA Global Optimization Method Based on Multi-Core CPU and Many-Core GPU

Guangyuan Kan,[1] Ke Liang,[2] Jiren Li,[1] Liuqian Ding,[1] Xiaoyan He,[1] Youbing Hu,[3] and Mark Amo-Boateng[2]

[1]*State Key Laboratory of Simulation and Regulation of Water Cycle in River Basin, Research Center on Flood & Drought Disaster Reduction of the Ministry of Water Resources, China Institute of Water Resources and Hydropower Research, Beijing 100038, China*
[2]*College of Hydrology and Water Resources, Hohai University, Nanjing 210098, China*
[3]*Hydrologic Bureau (Information Center) of the Huaihe River Commission, Bengbu 233001, China*

Correspondence should be addressed to Guangyuan Kan; kanguangyuan@126.com

Received 5 January 2016; Revised 17 February 2016; Accepted 2 March 2016

Academic Editor: Adel Hanna

The famous global optimization SCE-UA method, which has been widely used in the field of environmental model parameter calibration, is an effective and robust method. However, the SCE-UA method has a high computational load which prohibits the application of SCE-UA to high dimensional and complex problems. In recent years, the hardware of computer, such as multi-core CPUs and many-core GPUs, improves significantly. These much more powerful new hardware and their software ecosystems provide an opportunity to accelerate the SCE-UA method. In this paper, we proposed two parallel SCE-UA methods and implemented them on Intel multi-core CPU and NVIDIA many-core GPU by OpenMP and CUDA Fortran, respectively. The Griewank benchmark function was adopted in this paper to test and compare the performances of the serial and parallel SCE-UA methods. According to the results of the comparison, some useful advises were given to direct how to properly use the parallel SCE-UA methods.

1. Introduction

There are a large number of intelligent optimization algorithms in the field of parameter optimization of the environmental models, such as the genetic algorithm (GA) [1] and the particle swarm optimization (PSO) [2, 3]. Among these algorithms, the shuffled complex evolution method developed at The University of Arizona (SCE-UA) is recognized as an effective and robust global optimization technique for calibrating environmental models [4–9]. However, for problems with high dimensionality, complex objective function response surface, and high objective function computational load, it is resource-demanding and time-consuming. This disadvantage evokes the need for efficient acceleration of the SCE-UA method. The SCE-UA method is inherently parallel and should be accelerated at algorithm level. Besides, the parallel algorithm needs to be properly implemented on powerful parallel computation hardware. With the development of the parallel computing technology, the best way is the utilization of the heterogeneous computing system, which is composed by the multi-core central processing units (CPUs) and the many-core graphics processing units (GPUs). However, in previous literatures, the algorithm level parallelization analysis and the parallelization based on the heterogeneous computing system for the SCE-UA method are rare.

Parallel computing has been more and more popular in one form or another for many decades. In the early stages it was generally restricted to practitioners who had access to large and expensive machines. Today, things are quite different. Almost all consumer desktop and laptop computers have CPUs with multiple cores. Multi-core CPU hardware systems are build up on a set of processors which have access to a common memory. This architecture is recognized as shared-memory system. By placing several cores on a chip, multi-core processors offer a way to improve the performance of microprocessors. In the programming model of the multi-core processors, the parallelization is implemented by creating "threads" which represent separate tasks run by different

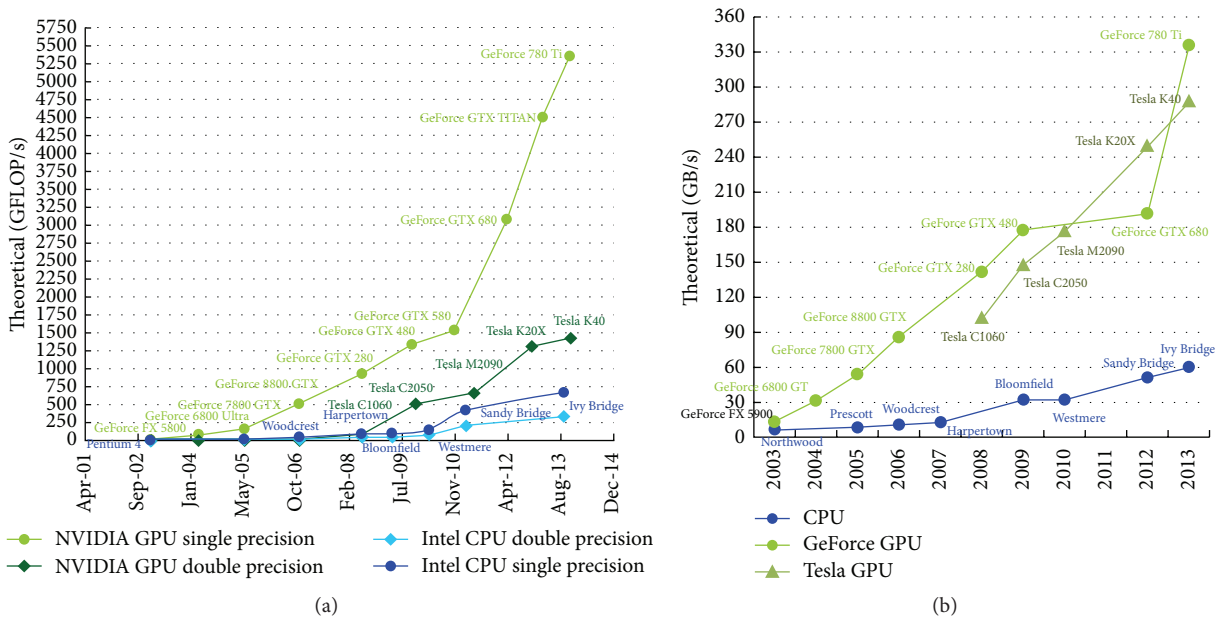

Figure 1: Comparison of computation capability of CPU and GPU: (a) floating-point operations per second (FLOP/s) for the CPU and GPU; (b) memory bandwidth for the CPU and GPU [13].

CPU cores. With multi-core CPUs, several existing or new programming models and environments can help users [10]. For example, OpenMP, Pthread, Cilk [11], and even MATLAB can be considered tools to help users implement programs on multi-core CPUs. Among those tools, OpenMP is adopted for the parallelization of the SCE-UA method on multi-core CPU systems owing to its simplicity and good efficiency.

Many-core GPU and multi-core CPU are two kinds of completely different hardware systems. The GPU is a highly parallel, multithreaded, many-core processor with tremendous computational horsepower and very high memory bandwidth, as illustrated by Figures 1(a) and 1(b) [12, 13]. NVIDIA introduced CUDA (compute unified device architecture), a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve complex computational problems. CUDA guides the programmer to partition the problem into coarse subproblems that can be solved independently in parallel by blocks of threads and each subproblem into finer pieces that can be solved cooperatively in parallel by all threads within the block [13]. Therefore, we adopted CUDA Fortran as the tool for the parallelization of the SCE-UA method on many-core NVIDIA GPU systems.

The objectives of this paper are the following. (1) Rare previous literature is about the parallelization and acceleration of the SCE-UA method. We analyze which part of the SCE-UA method could be parallelized. We redesigned and accelerated the SCE-UA method in the algorithm level and made the method highly suited to the multi-core CPU and many-core GPU. (2) The multi-core CPUs and many-core GPUs have not been applied for the parallelization and acceleration of the SCE-UA method previously. We implement parallel SCE-UA on these two kinds of hardware systems by utilizing the OpenMP and CUDA Fortran. (3) The Griewank benchmark function is used in this paper to test and compare the performances of the serial and parallel SCE-UA methods. According to the results of the comparison, some useful advises are given to direct how to properly use the parallel SCE-UA methods.

2. Methodology

2.1. The Serial SCE-UA Method (Serial-SCE-UA). The SCE-UA method is specifically designed to deal with the peculiarities encountered in environmental model calibration. The method is based on a synthesis of four concepts: (1) combination of deterministic and probabilistic approaches; (2) systematic evolution of a "complex" of points spanning the parameter space, in the direction of global improvement; (3) competitive evolution; (4) complex shuffling. The synthesis of these elements makes the SCE-UA method effective and robust, and also flexible and efficient. A detailed presentation of the theory underlying the SCE-UA algorithm could be found in Duan's papers [5, 14]. Duan provides MATLAB and Fortran 77 SCE-UA codes on his official website. These two versions are serial codes and are implemented on single core CPU. They can be recognized as the standard SCE-UA codes. In this paper, the serial SCE-UA CPU code is revised from Duan's MATLAB version and is implemented in Fortran 90. This serial SCE-UA CPU code is utilized as the base line for the performance comparisons.

2.2. The Parallel SCE-UA Method on Multi-Core CPUs (OMP-SCE-UA) and Many-Core GPUs (CUDA-SCE-UA)

2.2.1. Overall Description. In this study, we proposed a parallel SCE-UA method and implement it on multi-core CPU

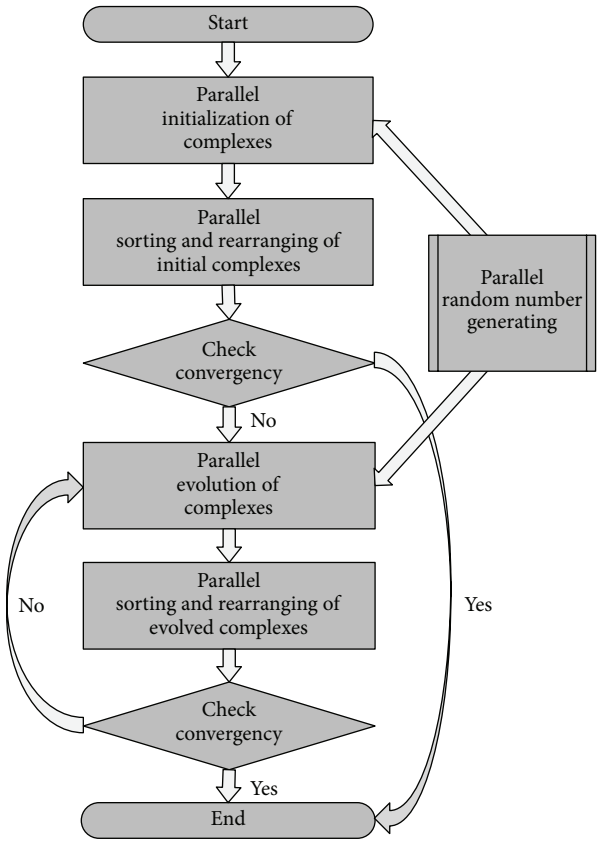

Figure 2: Flowchart of the parallel SCE-UA method.

by OpenMP and many-core GPU by CUDA Fortran, respectively. The SCE-UA method is inherently parallel and should be redesigned to implement the parallelization. According to the computation flow path of the SCE-UA algorithm, the following parts of the SCE-UA method are parallelizable: (1) initialization of complexes, (2) evolution of complexes, (3) random number generating, and (4) sorting and rearranging of complexes. The flowchart of the parallel SCE-UA method is shown in Figure 2.

2.2.2. Parallel Initialization of Complexes. Before the evolution of complexes, the SCE-UA method randomly generates a set of initial points to constitute the initial complexes where each point represents a potential solution for the problem. The generation steps include the generation of initial points and the computation of objective function value of each point. The generation of initial points can be parallelized and requires a parallel random number generator which will be described in Section 2.2.4. The computation of objective function value for each initial point is unrelated to other points and can also be parallelized. Supposing we need to generate *npt* initial points and their corresponding objective function values, we create *npt* threads on the CPU or GPU. In each thread, we compute the objective function value for the corresponding randomly generated point. By using multi-core CPU and many-core GPU, the *npt* threads can be executed in parallel to obtain the objective function values.

2.2.3. Parallel Evolution of Complexes. The evolution of complexes improves the objective function values by evolving *ngs* complexes towards the global optimum. The process of complex evolution is inherently parallel and should be parallelized. For each complex evolution loop, we create *ngs* threads to represent *ngs* complexes and evolve these threads in parallel on the CPU or GPU to implement the parallel evolution. When the stopping criterion is satisfied, the complex evolution is stopped. In each complex evolution loop, for each complex, we perform the CCE (complex competitive evolution) and the points sequence rearranging for *nspl* steps. The CCE contains three typical operations to imitate the genetic algorithm and the downhill simplex algorithm, which includes simplex choosing (the selection), the worst point reflection (the crossover), and the reflection failed point random regeneration (the mutation). In the mutation step, we need a parallel random number generator which is described in Section 2.2.4. The points sequence rearranging utilizes the quick sort algorithm to sort the evolved points in increasing objective function values and rearranges the point sequence according to the objective function values. After that, the algorithm is ready for the next round CCE operation.

2.2.4. Parallel Random Number Generating. During the initialization and evolution of complexes, we need to generate uniformly distributed random numbers for each thread. Because the initialization and evolution process is parallel, the random number generating process should be run in parallel. We design the following parallel random number generating method:

(1) For each thread, we generate a random number sequence, respectively [15–17]. This method promises that the generation process for each thread is unrelated to other threads and guarantees the high quality of the generated random number sequence.

(2) In order to avoid the correlation between different random number sequences and obtain better random characteristics, we adopt different randomly generated seeds for each thread and utilize the Mersenne twister random number generator instead of the widely used linear congruential generator [18, 19]. For each thread, a separately generated random seed is adopted and catered for the Mersenne twister random number generator to generate a random number sequence for the corresponding thread.

2.2.5. Parallel Sorting and Rearranging of Complexes. After the initialization or the shuffling evolution loop, the initial or evolved complexes should be sorted in increasing objective function values and rearranged according to their corresponding objective function values. Because the sorting and rearranging processes are inherently parallel, we should parallelize these processes by using the radix parallel sorting method [20, 21] and the parallel rearranging method. The parallel sorting and rearranging are also implemented on the multi-core CPU and the many-core GPU.

Table 1: Total execution time (seconds) of the Serial-SCE-UA.

nopt	*ngs*								
	4	8	16	32	64	128	256	512	1024
10	43.95	96.14	185.58	358.38	719.08	1388.17	2804.75	5688.64	11225.29
20	79.00	162.72	320.67	619.09	1201.86	2441.35	4880.35	9849.35	19492.42
30	**130.04**	**252.29**	**484.13**	**919.92**	**1850.08**	**3697.84**	**7207.62**	**14263.88**	**28609.81**
40	193.28	360.20	670.29	1238.20	2505.45	4777.02	9627.71	19040.44	37697.98
50	243.67	420.27	833.65	1580.19	3081.36	6235.98	12190.24	23760.53	46559.85

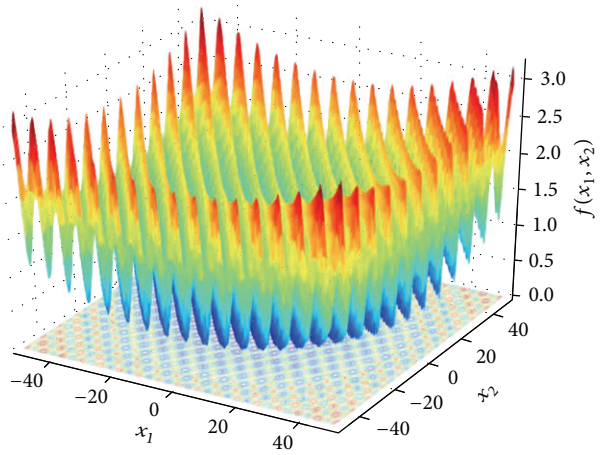

Figure 3: The two-dimensional Griewank function.

2.3. Experimental Studies of the Serial and Parallel SCE-UA

2.3.1. The Griewank Benchmark Function. The performance comparison of serial and parallel SCE-UA methods is based on the Griewank benchmark function. The Griewank global optimization problem is a multimodal minimization problem defined as follows:

$$f(x_1, x_2, \ldots, x_n) = \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1. \quad (1)$$

Here, n represents the number of dimensions (i.e., the number of decision variables) and $x_i \in [-100, 100]$ for $i = 1, 2, \ldots, n$. The global optimum of the Griewank problem is $f(x_1, x_2, \ldots, x_n) = 0$ for $x_i = 0$ for $i = 1, 2, \ldots, n$. As a simple example, the two-dimensional Griewank function is demonstrated in Figure 3. The Griewank problem is complex enough to test the global property of the SCE-UA and the dimension can be changed to test the performance of the SCE-UA. Therefore, we adopt it as the benchmark function in this study.

2.3.2. Settings for the Hardware and Software Utilized in This Study

(1) The Hardware Utilized in This Study. In this study, we utilized the Intel Core i7-4710HQ CPU with hyperthreading (4 CPU cores with 8 threads) and the NVIDIA Geforce GTX 850M (DDR3 version) (see Figure 4). We can see from Figure 4 that the i7-4710HQ CPU has 8 logical CPU cores and the GTX 850M GPU has 640 CUDA GPU cores, which means the CPU and GPU can make full use of their computation capability by using 8 and 640 threads in parallel, respectively.

(2) Software Settings for This Study. The SCE-UA method has several algorithm parameters which control the convergence behavior of the algorithm. They are *maxn*, maximum number of objective function trials allowed before optimization is terminated; *kstop*, number of shuffling loops in which the objective function must improve by the specified percentage *pcento* or else optimization will be terminated; *pcento*, percentage by which the objective function must change in the specified number of shuffling loops *kstop* or else the optimization is terminated; *peps*, minimum parameter space allowed before optimization is terminated. For the purpose of fair comparison, we set the parameters as follows: *maxn* = positive infinity; *kstop* = 5; *pcento* = 0.1; *peps* = 0.000001.

In order to analyze the performance of the serial and parallel SCE-UA algorithm, we need to adjust settings for the serial and parallel algorithm to test how the algorithm performs. These settings are *nopt*, number of decision variables; *ngs*, number of complexes; *nobj*, loop number which is used to test the objective function computation overhead (each loop contains four floating-point arithmetic operations, i.e., including one addition, one subtraction, one multiplication, and one division. Larger *nobj* corresponds to higher computation overhead). There is one thing that must be noted for the purpose of fair comparison, the time consumed by memory allocation on GPU, transfer between CPU and GPU, and deallocation from GPU that is also considered in this research. All the SCE-UA methods are implemented in single floating-point precision.

3. Results and Discussion

3.1. Performance Comparison Based on Total Execution Time. In this section, we test the performances of the serial and parallel SCE-UA methods based on the total execution time (in seconds). For the purpose of fair comparison, we set the *nobj* = 1000000 for these comparisons. We adjust the *nopt* and *ngs* to test how the algorithm performs.

3.1.1. Serial-SCE-UA. The total execution time of the serial SCE-UA is demonstrated in Figure 5 and Table 1. As we can see, with the increasing of the *nopt*, the total execution time increases. As for the increasing of the *ngs*, the execution time varies from 43.95 s to 46559.85 s in proportion. Let *nopt* = 30 as an example (the bolded line of Table 1); the *ngs*

Table 2: Total execution time (seconds) of the OMP-SCE-UA.

nopt	*ngs*								
	4	8	16	32	64	128	256	512	1024
10	13.32	14.96	26.51	50.48	96.56	189.85	366.68	729.57	1451.04
20	23.63	24.60	43.21	81.76	162.38	321.86	638.98	1270.75	2530.29
30	**34.02**	**36.80**	**65.99**	**125.92**	**243.61**	**474.52**	**944.30**	**1877.73**	**3737.83**
40	49.39	47.56	88.61	165.91	337.02	641.63	1278.80	2488.18	4945.17
50	74.37	63.01	112.41	210.94	420.48	798.60	1595.91	3177.03	6186.20

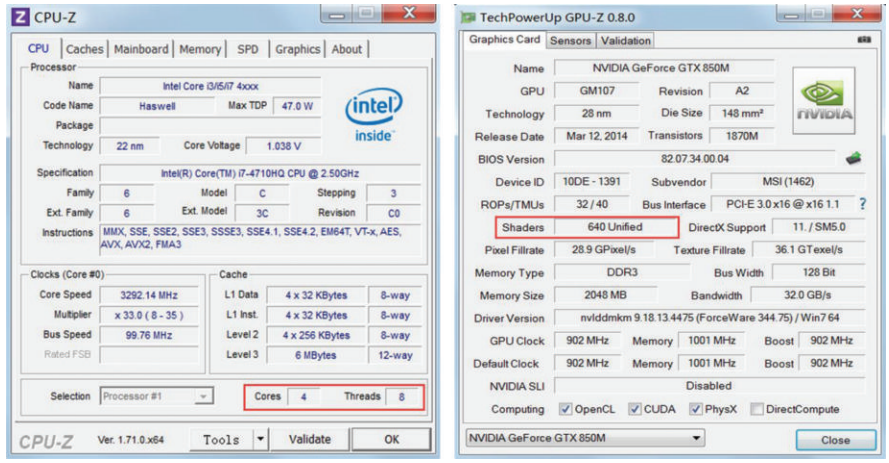

Figure 4: The CPU and GPU used in this study.

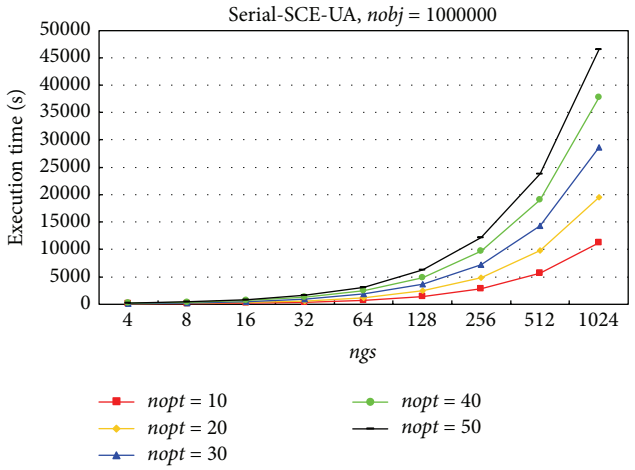

Figure 5: Total execution time (seconds) of the Serial-SCE-UA.

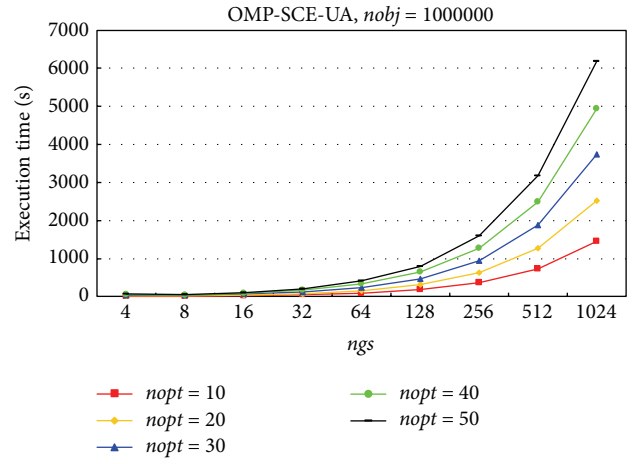

Figure 6: Total execution time (seconds) of the OMP-SCE-UA.

increases from 4 to 1024 with a multiple of two. We can observe that the total execution time increases from 130.04 s to 28609.81 s with approximately the same multiple of two. This is because the serial version only utilizes one CPU core; with the increasing of *ngs*, the total execution time of course increases with approximately the same proportion. For *ngs* < 16, the execution time is not doubled. This is mainly because of less computational overhead; the CPU resources used for the thread dispatch are relatively higher than the computation. The dispatch of threads cost relatively more time which makes the execution time not doubled.

3.1.2. OMP-SCE-UA. The total execution time of the OpenMP SCE-UA is demonstrated in Figure 6 and Table 2. As we can see, with the increasing of the *nopt*, the total execution time increases. As for the increasing of the *ngs*, the total execution time varies from 13.32 s to 6186.20 s. The time consumed by the OpenMP version is much less than the serial version. Different from the serial version, with the increasing of the *ngs* and when the *ngs* is smaller than 16, the total execution time increases not according to the same increasing multiple of the *ngs*. Let *nopt* = 30 as an example (the bolded line of Table 2); the *ngs* increases from 4 to 1024 with a multiple of

Table 3: Total execution time (seconds) of the CUDA-SCE-UA.

nopt	*ngs*								
	4	8	16	32	64	128	256	512	1024
10	224.62	228.95	241.62	251.60	245.62	247.45	260.40	264.61	267.10
20	406.99	412.23	409.77	412.14	414.13	396.97	416.59	420.15	424.20
30	670.82	648.42	604.67	626.22	629.26	611.52	619.27	623.42	624.14
40	980.73	937.47	886.81	860.83	805.71	814.37	814.33	818.35	839.48
50	1400.51	1173.31	1081.71	1078.16	1115.32	1046.00	1080.26	1053.03	1064.78

Table 4: Speed-up ratio of the OMP-SCE-UA versus the Serial-SCE-UA.

nopt	*ngs*								
	4	8	16	32	64	128	256	512	1024
10	3.30	6.43	7.00	7.10	7.45	7.31	7.65	7.80	7.74
20	3.34	6.61	7.42	7.57	7.40	7.59	7.64	7.75	7.70
30	3.82	6.86	7.34	7.31	7.59	7.79	7.63	7.60	7.65
40	3.91	7.57	7.56	7.46	7.43	7.45	7.53	7.65	7.62
50	3.28	6.67	7.42	7.49	7.33	7.81	7.64	7.48	7.53

Table 5: Speed-up ratio of the CUDA-SCE-UA versus the Serial-SCE-UA.

nopt	*ngs*								
	4	8	16	32	64	128	256	512	1024
10	0.20	0.42	0.77	1.42	2.93	5.61	10.77	21.50	42.03
20	0.19	0.39	0.78	1.50	2.90	6.15	11.72	23.44	45.95
30	0.19	0.39	0.80	1.47	2.94	6.05	11.64	22.88	45.84
40	0.20	0.38	0.76	1.44	3.11	5.87	11.82	23.27	44.91
50	0.17	0.36	0.77	1.47	2.76	5.96	11.28	22.56	43.73

two. We can observe that the total execution time increases from 34.02 s to 3737.83 s. When *ngs* is smaller than 16, the increasing multiple of total execution time is not two (less than two). When *ngs* is equal to or larger than 16, the increasing multiple is approximately equal to two. This is because the OpenMP version can utilize at most 8 CPU cores (the CPU used in this study has at most 8 cores). When the *ngs* is less than 16, the number of threads equals the CPU cores used. Larger *ngs* requires more CPU cores and the CPU can provide enough cores for the OMP-SCE-UA (when *ngs* is less than 16) to boost the performance. Therefore, the increasing multiple of total execution time is not two (less than two). However, when the *ngs* is equal to or larger than 16, with the increasing of *ngs*, the OMP-SCE-UA requires more CPU cores to boost the performance. However, there are only 8 CPU cores and the CPU cannot provide more computation resources for the OMP-SCE-UA to boost the performance. Therefore, the execution time increases with the same increasing multiple of the *ngs* (when *ngs* is equal to or larger than 16).

3.1.3. CUDA-SCE-UA. The total execution time of the CUDA SCE-UA is demonstrated in Figure 7 and Table 3. As we can see, with the increasing of the *nopt*, the total execution time increases. As for the increasing of the *ngs*, the total execution time varies from 224.62 s to 1400.51 s. The performance of the CUDA version is completely different from the serial and OpenMP version. With the increasing of the *ngs*, the execution time changes little and decreases a little. This is because

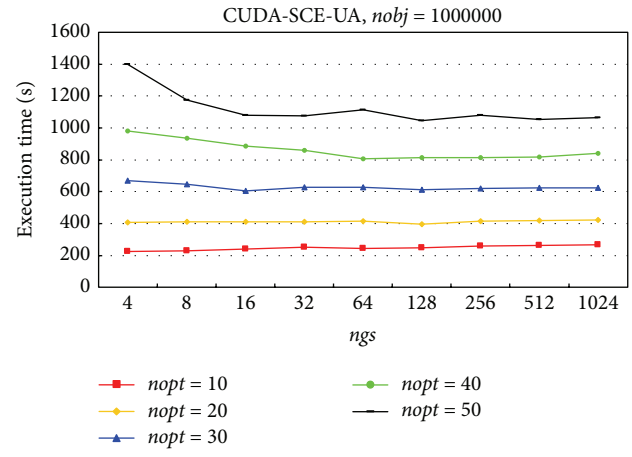

Figure 7: Total execution time (seconds) of the CUDA-SCE-UA.

with less *ngs* (i.e., less GPU threads) the GPU cannot hide the latency of the global memory access. The latency slows down the performance of the algorithm. With larger *ngs*, the GPU can hide the latency by launching many threads simultaneously and therefore boost the performance of the algorithm.

3.2. Speedup Ratio Analysis. The speedup ratio statistics are demonstrated in Figure 8 and Tables 4 and 5. As shown in figure and tables, the speedup ratio of the OpenMP version

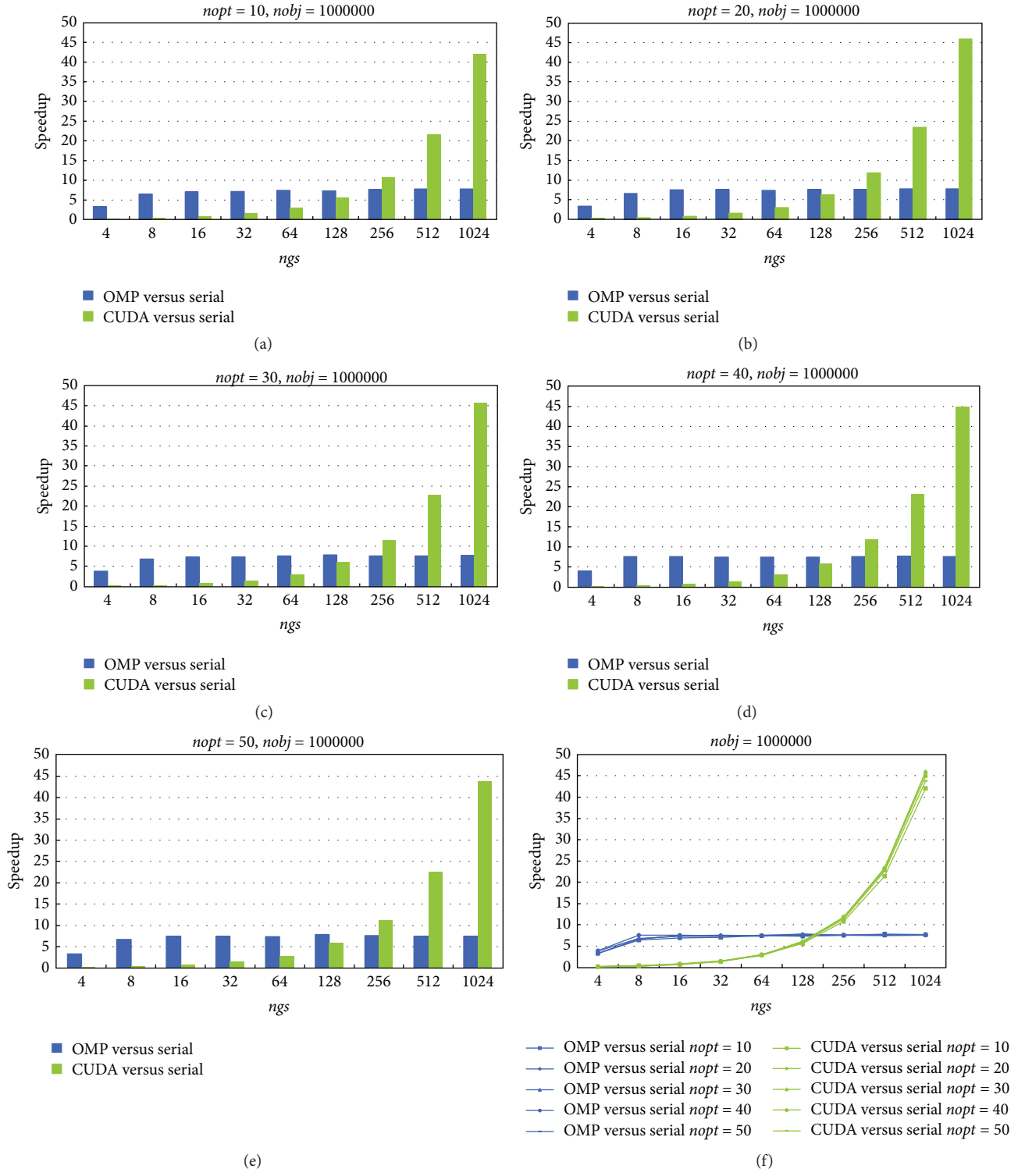

Figure 8: Speedup ratio of the parallel SCE-UA versus the serial SCE-UA.

varies from 3.28x to 7.81x, and the speedup ratio of the CUDA version varies from 0.17x to 45.95x. The speedup ratio of the OpenMP version is all less than 8x. The reason is as follows. The OpenMP program creates as many threads as possible to make full use of all the CPU cores. When the number of threads becomes larger than 8, because the CPU utilized in this study has only 8 CPU cores, the creation, management, dispatching, and destroying of CPU threads consume more CPU resources and prevent the speedup ratio from being larger than 8x. As for the CUDA version, when *ngs* (that equals the number of threads created by the GPU) is small, the speedup ratio of the CUDA version is less than

the OpenMP version. This is because the clock speed of the CPU core is 2.5 GHz which is much higher than the clock speed of the GPU core (902 MHz). However, when *ngs* is very large (*ngs* > 128), the speedup ratio of the CUDA version becomes much higher than the OpenMP version. This is because the GPU has much more cores (640 CUDA GPU cores) than the CPU (only 8 CPU cores) and can launch much more parallel threads to boost the performance. As for the OpenMP version, the speedup ratio nearly reaches maximum value when *ngs* equals 8 and 16, and the speedup ratio cannot become larger by increasing the *ngs*. As for the CUDA version, the speedup ratio can become very large by increasing the *ngs* and the performance becomes much more satisfactory than the OpenMP version. At last, we can observe from Figure 8(f) that the *nopt* (i.e., the number of decision variables) has very little impact on the speedup ratio. This means that although the total execution time increases with the increasing of the *nopt*, the speedup ratio has very little relationship with the *nopt*.

3.3. Analysis of the Impact of the Objective Function Computational Overhead. The relationship between the *nobj* and the speedup ratio is demonstrated in Figure 9. We set *nopt* = 20. The *ngs* varies from 4 to 1024. We can see that with the increasing of the *nobj* (i.e., the increasing of the objective function computational overhead), the speedup ratio becomes larger. There is one difference between the OpenMP and the CUDA version. With the increasing of the *ngs* and the *nobj*, the optimization problem becomes more complex and the computational overhead becomes heavier. The increasing extent of the speedup ratio of the OpenMP version is not very wide, and the speedup ratio is not higher than 8x. With the increasing of the *ngs*, the speedup ratio of the CUDA version becomes much higher than the OpenMP version, up to approximately 45x. These results show that the CUDA version performs much better than the OpenMP version under the condition of solving the problem with complex and high computational load objective function.

3.4. Optimization Accuracy Comparison. After checking the optimization results, we found that all the optimization problems converge to the global optimum. This fact shows that the parallel SCE-UA can find the global optimum with the same accuracy as the serial SCE-UA.

3.5. Some Useful Advices on How to Properly Utilize the Parallel SCE-UA Method. After carefully checking the results, we give some useful advices on how to properly utilize the parallel SCE-UA methods:

(1) Both of the serial and parallel versions can give the correct optimization result. Therefore, for normal problems, the serial and parallel versions are both applicable and can converge to the global optimum. For simple problems (*ngs* < 256), the serial and parallel versions are all good to use but the parallel version may not obtain satisfactory speedup ratio because the overhead of creating, dispatching, and destroying of the threads is usually higher than the computation overhead of the optimization algorithm. Therefore, for simple problems (*ngs* < 256), we recommend using the serial version.

(2) The parallel version runs faster than the serial version especially for complex (*ngs* > 128) and high dimensional problems (*nopt* > 40). Therefore, we recommend using the parallel version for these problems. For complex problem with relatively small *ngs* (*ngs* < 256), we recommend using the OpenMP version to obtain a better performance. For complex problem with large *ngs* (*ngs* > 128), the CUDA version is a better choice. As for problems with very high objective function computational load, we recommend to use the CUDA version.

(3) We should announce that the parallel version needs more memory than the serial version. This is because the parallel version needs to create arrays and vectors separately for each thread to ensure the correctness of parallel execution. Therefore, for very complex and very high dimensional problems, the parallel version may need much more memory and cause the memory overflow. We recommend using the 64-bit exe on 64-bit operating systems to cope with these kinds of problems because that 64-bit program can utilize more memory than the 32-bit program. The dimensionality of the problem (*nopt*) do not affect the speedup ratio very much; the only constraint is the memory size. Larger *nopt* need more memory to store the large and complex population. Therefore, for these problems we recommend installing more CPU memory for the OpenMP version or adopting the Tesla GPU card (usually has more GPU memory) for the CUDA version to provide more memory to ensure the successful execution of the optimization.

4. Conclusions

In this paper, we proposed parallel SCE-UA method and implemented it on Intel multi-core CPU and NVIDIA many-core GPU by OpenMP and CUDA Fortran. The serial and parallel SCE-UA codes were optimized at the same level to ensure a fair comparison. The Griewank benchmark function was adopted in this paper to test and compare the performances of the serial and parallel SCE-UA methods. According to the results of the comparison, some useful advises were given to direct how to properly use the parallel SCE-UA. Three conclusions can be stated here:

(1) Both of the serial and parallel versions can obtain the global optimum with a satisfactory probability. We can now produce reliable estimates of global optima for large complex optimization problems by both the serial and parallel versions of the SCE-UA methods.

(2) The experimental studies were carried out by using the Griewank benchmark function. This benchmark function embodies many typical problems encountered in the global optimization. Therefore, the recommendations for the parallel SCE-UA derived

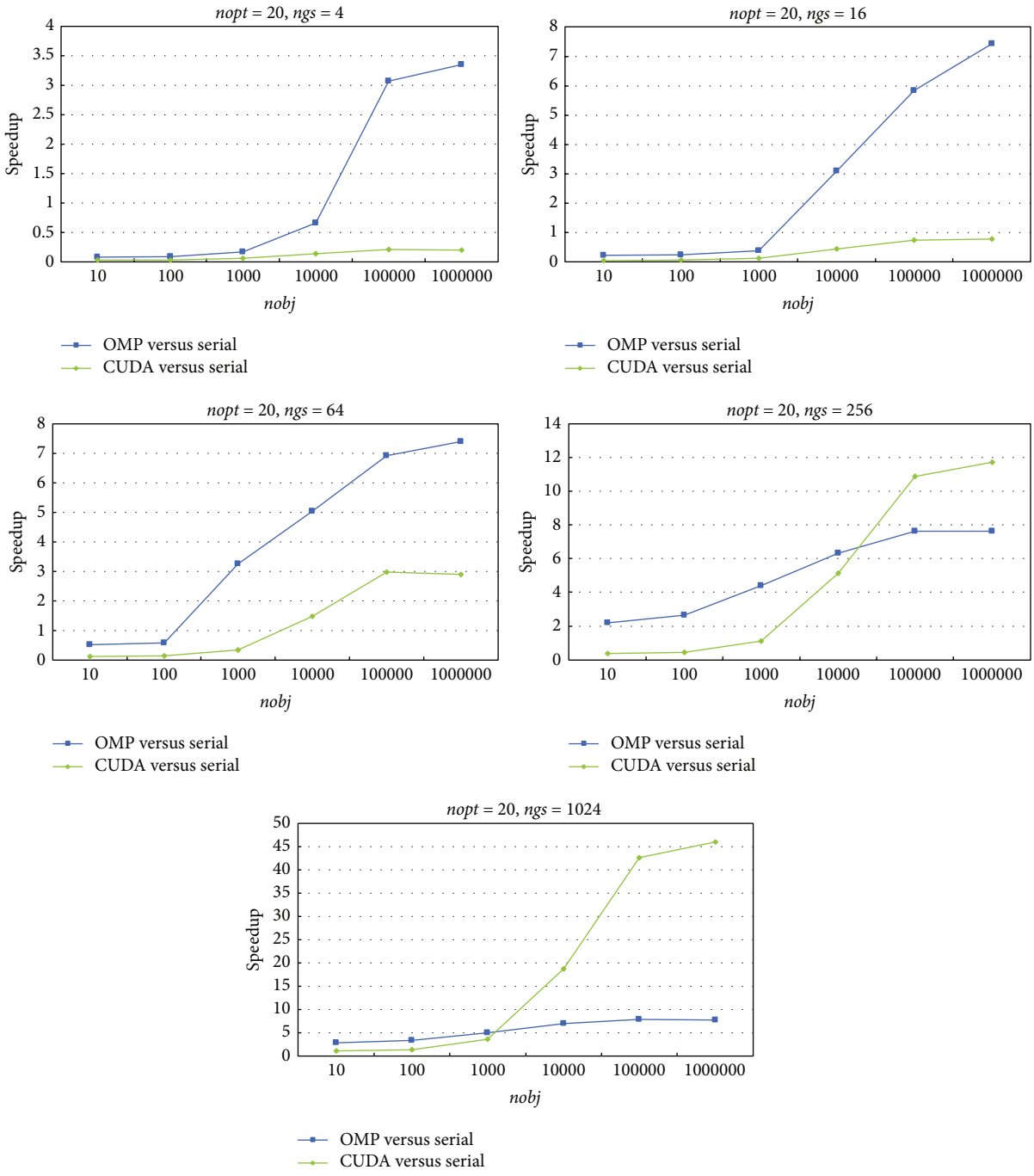

Figure 9: Relationship between the *nobj* and the speedup ratio.

here can be recognized as guidelines for most applications.

(3) With the advent of the newly developed parallel SCE-UA methods, we can now produce reliable and much faster estimates of global optima for large complex optimization problems by using the parallel SCE-UA methods. The OpenMP version is recommended to be used on medium problems and the CUDA version is recommended to be used on large problems.

Competing Interests

The authors declare that they have no competing interests.

Acknowledgments

This research was funded by the IWHR Scientific Research Projects of Outstanding Young Scientists "Research and Application on the Fast Global Optimization Method for the Xinanjiang Model Parameters Based on the High

Performance Heterogeneous Computing" (no. KY1605, JZ0145B052016), Specific Research of China Institute of Water Resources and Hydropower Research (Grant no. Fangji 1240), the Third Sub-Project: Flood Forecasting, Controlling and Flood Prevention Aided Software Development-Flood Control Early Warning Communication System and Flood Forecasting, Controlling and Flood Prevention Aided Software Development for Poyang Lake Area of Jiangxi Province (0628-136006104242, JZ0205A432013, and SLXMB200902), the NNSF of China, Numerical Simulation Technology of Flash Flood Based on Godunov Scheme and Its Mechanism Study by Experiment (no. 51509263), the NNSF of China, Study on the Integrated Assessment Model for Risk and Benefit of Dynamic Control of Reservoir Water Level in Flood Season (no. 51509268), and the NNSF of China, Estimation of Regional Evapotranspiration Using Remotely Sensed Data Based on the Theoretical VFC/LST Trapezoid Space (no. 41501415). The authors gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research.

References

[1] J. H. Holland, *Adaptation in Natural and Artificial Systems*, MIT Press, Cambridge, Mass, USA, 1992.

[2] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of IEEE International Conference on Neural Networks*, pp. 1942–1948, Perth, Australia, December 1995.

[3] Y. Shi and R. C. Eberhart, "Modified particle swarm optimizer," in *Proceedings of the IEEE International Conference on Evolutionary Computation (ICEC '98)*, pp. 69–73, May 1998.

[4] S. Sorooshian, Q. Duan, and V. K. Gupta, "Calibration of rainfall-runoff models: application of global optimization to the Sacramento soil moisture accounting model," *Water Resources Research*, vol. 29, no. 4, pp. 1185–1194, 1993.

[5] Q. Duan, S. Sorooshian, and V. K. Gupta, "Effective and efficient global optimization for conceptual rainfall-runoff models," *Water Resources Research*, vol. 28, no. 4, pp. 1015–1031, 1992.

[6] Q. Duan, S. Sorooshian, and V. K. Gupta, "Optimal use of the SCE-UA global optimization method for calibrating watershed models," *Journal of Hydrology*, vol. 158, no. 3-4, pp. 265–284, 1994.

[7] P. O. Yapo, H. V. Gupta, and S. Sorooshian, "Automatic calibration of conceptual rainfall-runoff models: sensitivity to calibration data," *Journal of Hydrology*, vol. 181, no. 1–4, pp. 23–48, 1996.

[8] V. A. Cooper, V. T. V. Nguyen, and J. A. Nicell, "Evaluation of global optimization methods for conceptual rainfall-runoff model calibration," *Water Science and Technology*, vol. 36, no. 5, pp. 53–60, 1997.

[9] K. Eckhardt and J. G. Arnold, "Automatic calibration of a distributed catchment model," *Journal of Hydrology*, vol. 251, no. 1-2, pp. 103–109, 2001.

[10] L. Zheng, Y. Lu, M. Guo, S. Guo, and C.-Z. Xu, "Architecture-based design and optimization of genetic algorithms on multi- and many-core systems," *Future Generation Computer Systems*, vol. 38, pp. 75–91, 2014.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, 1996.

[12] G. Ruetsch and M. Fatica, *CUDA Fortran for Scientists and Engineers: Best Practices for Efficient CUDA Fortran Programming*, Morgan Kaufmann, 2013.

[13] NVIDIA, *CUDA C Programming Guide: Design Guide*, PG-02829-001_v6.5, NVIDIA, Santa Clara, Calif, USA, 2014.

[14] Q. Y. Duan, V. K. Gupta, and S. Sorooshian, "Shuffled complex evolution approach for effective and efficient global minimization," *Journal of Optimization Theory and Applications*, vol. 76, no. 3, pp. 501–521, 1993.

[15] L. Y. Barash and L. N. Shchur, "RNGSSELIB: program library for random number generation. More generators, parallel streams of random numbers and Fortran compatibility," *Computer Physics Communications*, vol. 184, no. 10, pp. 2367–2369, 2013.

[16] S. Gao and G. D. Peterson, "GASPRNG: GPU accelerated scalable parallel random number generator library," *Computer Physics Communications*, vol. 184, no. 4, pp. 1241–1249, 2013.

[17] K. G. Savvidy, "The MIXMAX random number generator," *Computer Physics Communications*, vol. 196, pp. 161–165, 2015.

[18] V. Demchik, "Pseudo-random number generators for Monte Carlo simulations on ATI graphics processing units," *Computer Physics Communications*, vol. 182, no. 3, pp. 692–705, 2011.

[19] S. Harase, "On the $\mathbb{F}_2$-linear relations of Mersenne twister pseudorandom number generators," *Mathematics and Computers in Simulation*, vol. 100, pp. 103–113, 2014.

[20] S.-J. Lee, M. Jeon, D. Kim, and A. Sohn, "Partitioned parallel radix sort," *Journal of Parallel and Distributed Computing*, vol. 62, no. 4, pp. 656–668, 2002.

[21] T. Thanakulwarapas and J. Werapun, "An optimized bitonic sorting strategy with midpoint-based dynamic communication," *Journal of Parallel and Distributed Computing*, vol. 84, pp. 37–50, 2015.