A heterogeneous computing accelerated SCE-UA global optimization method using OpenMP, OpenCL, CUDA, and OpenACC

Guangyuan Kan, Xiaoyan He, Liuqian Ding, Jiren Li, Ke Liang and Yang Hong

ABSTRACT

The shuffled complex evolution optimization developed at the University of Arizona (SCE-UA) has been successfully applied in various kinds of scientific and engineering optimization applications, such as hydrological model parameter calibration, for many years. The algorithm possesses good global optimality, convergence stability and robustness. However, benchmark and real-world applications reveal the poor computational efficiency of the SCE-UA. This research aims at the parallelization and acceleration of the SCE-UA method based on powerful heterogeneous computing technology. The parallel SCE-UA is implemented on Intel Xeon multi-core CPU (by using OpenMP and OpenCL) and NVIDIA Tesla many-core GPU (by using OpenCL, CUDA, and OpenACC). The serial and parallel SCE-UA were tested based on the Griewank benchmark function. Comparison results indicate the parallel SCE-UA significantly improves computational efficiency compared to the original serial version. The OpenCL implementation obtains the best overall acceleration results however, with the most complex source code. The parallel SCE-UA has bright prospects to be applied in real-world applications.

Key words | CUDA, heterogeneous computing, OpenACC, OpenCL, OpenMP, SCE-UA

Guangyuan Kan (corresponding author)
Xiaoyan He
Liuqian Ding
Jiren Li
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
E-mail: *kanguangyuan@126.com*

Guangyuan Kan
Yang Hong
State Key Laboratory of Hydroscience and Engineering, Department of Hydraulic Engineering,
Tsinghua University,
Beijing 100084, China

Ke Liang
College of Hydrology and Water Resources,
Hohai University,
Nanjing 210098, China

Yang Hong
Department of Civil Engineering and Environmental Science,
University of Oklahoma,
Norman, OK, USA

INTRODUCTION

Global optimization is a hot but difficult issue in scientific research and engineering applications, such as hydrological model parameter optimization. It has been widely recognized that the achievement of a stable global optimum for highly complex benchmarks and real-world applications is still challenging. Issues, such as local minimums, high dimensionality, and premature convergence, etc., have become major obstacles for appropriate solution of optimization problems. To overcome these obstacles, many optimization methods and algorithms have been proposed. These include grid search, Monte Carlo random search, genetic algorithm, particle swarm optimization, SCE-UA (shuffled complex evolution optimization developed at the University of Arizona), etc. A huge number of optimization methods have been developed and applied in various kinds of scientific research and engineering applications; however, only a small number of methods perform satisfactorily in highly complex and nonlinear real-world applications such as hydrological model parameter optimization. Among these methods and algorithms, SCE-UA has been widely accepted and used in scientific and engineering optimization applications, especially for hydrological model parameter global calibration (Duan 1991; Li *et al.* 2014, 2016; Dong *et al.* 2015; Kan *et al.* 2015a, 2015b, 2016a, 2016b, 2016c, 2016d, 2016e; Zuo *et al.* 2016). It features good global optimality, stable robustness, and excellent consistency. Therefore, the SCE-UA method has been recognized as an effective and robust tool for solving the above-mentioned obstacles, to some extent.

doi: 10.2166/wst.2017.322

Even though SCE-UA has achieved great success in many fields, it still faces a severe computational efficiency issue when the optimization problem is highly complex. It needs to carry out large numbers of objective function evaluations during the evolution process. Taking the hydrological model parameter optimization as an example, it needs to repeatedly run the hydrological model to obtain the objective function values in each iteration. The computational burden is very high. It usually takes days or even weeks to finish the optimization task and this situation is not acceptable. The SCE-UA method is usually implemented by using serial programming and computing technology, such as serial Fortran, C/C++, and MATLAB, etc. For complex optimization problems such as hydrological model parameter automatic calibration, the computational load is very high and the execution speed of the serial code is too slow. The computational efficiency of the SCE-UA needs to be improved and researchers have made some efforts to improve it. There are mainly two ways: one is the improvement of the algorithm itself (Wang *et al.* 2014; Gong *et al.* 2015, 2016; Zhang *et al.* 2017) and the other is acceleration by using parallel computing supported new generation multi-core CPUs (central processing units) and many-core GPUs (graphics processing units) (Sharma *et al.* 2006; Muttil *et al.* 2007; Kan *et al.* 2016a, 2016b, 2016c, 2016d, 2016e). For the improvement of the algorithm itself it is necessary to introduce an artificial neural network or other data-driven surrogate model to mimic the objective function response surface. The selection of the surrogate model may affect the optimization accuracy and this operation has the possibility of optimization accuracy deterioration. The improvement of computational efficiency by using surrogate models has a computational efficiency upper limit which is limited by the specifically selected surrogate model. If users want to further improve the computational efficiency they have to select simpler surrogate models and this may lead to the deterioration of optimization accuracy. Acceleration by using highly parallel computing hardware, such as multi-core CPUs and many core GPUs, does not face the above-mentioned problems. It is based on algorithm parallelization and parallel programming technology. It can achieve satisfactory speedup ratio without loss of optimization accuracy. The only cost is the parallel algorithm implementation by using new programming techniques such as OpenMP and CUDA.

The acceleration of the SCE-UA by using multi-core CPUs and many-core GPUs based on OpenMP and CUDA has been proposed and tested in previous literature. However, there are many popular parallel programming techniques, such as OpenMP, OpenCL, CUDA, and OpenACC, which can be applied to the implementation of the parallel SCE-UA. Performances and code complexities of different implementations may vary significantly. To achieve effective and efficient use of the parallel SCE-UA, it is of great importance to make comparisons and give some useful guidelines for the selection and utilization of different parallel programing techniques. Unfortunately, further investigations of the parallel SCE-UA method, such as parallel algorithm implementations by using different popular parallel programming techniques, like OpenMP, OpenCL, CUDA, and OpenACC, and performance comparisons, have not been carried out. To carry out more in-depth research on the parallel SCE-UA and compare algorithm performances and code complexities of different implementations, we proposed a parallel SCE-UA method and implemented it on an Intel Xeon multi-core CPU and NVIDIA Tesla many-core GPU by using OpenMP, OpenCL, CUDA (NVIDIA Corporation 2015), and OpenACC. Performance tests of the serial and parallel SCE-UA were carried out based on the Griewank benchmark function to test the execution speed, optimization accuracy, robustness, and consistency. The source codes of OpenMP, OpenCL, CUDA, and OpenACC implementations were also analyzed to study the relationship between the speedup ratio and source code complexity. Some useful guidelines are given to potential users of parallel SCE-UA to help them achieve better acceleration results.

SERIAL SCE-UA

The original serial SCE-UA 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 (Holland 1975); and (4) complex shuffling (Duan *et al.* 1992, 1993; Sorooshian *et al.* 1993). A general description of the steps of the SCE-UA is given below (a more detailed presentation can be found in Duan *et al.* (1992, 1993)).

(1) Generate sample. Sample s points randomly in the feasible parameter space and compute the criterion value at each point. In the absence of prior information on the approximate location of the global optimum, use a uniform probability distribution.
(2) Rank points. Sort the s points in order of increasing criterion value so that the first point represents the smallest

criterion value (assuming that the goal is to minimize the criterion value).

(3) Partition into complexes. Partition the s points into p complexes, each containing m points. The complexes are partitioned such that the first complex contains every $p(k-1)+1$ ranked point, the second complex contains every $p(k-1)+2$ ranked point, and so on, where $k=1, 2, \ldots, m$.

(4) Evolve each complex. Evolve each complex according to the competitive complex evolution (CCE) algorithm (which is elaborated upon below).

(5) Shuffle complexes. Combine the points in the evolved complexes into a a single sample population; sort the sample population in order of increasing criterion value; re-partition the sample population into p complexes according to the procedure specified in Step 3.

(6) Check convergence. If any of the convergence criteria (stated below) are satisfied, stop; otherwise, return to Step 4.

The SCE-UA method iterates to converge towards the global optimal point. Therefore, it should be stopped by using some termination criteria to indicate the arrival of the optimal point. The criteria discussed below were used in this research.

(a) Objective function convergence: The objective function convergence indicates that the search should be stopped when the algorithm is unable to further improve the objective function value over a pre-specified number of iterations (K_{stop}). This is implemented by:

$$\left|\frac{f_i - f_{i-1}}{f_i}\right| \leq Tolerance_{obj}$$

where f_i and $f_{i\text{-}1}$ denote the objective function values at the ith and i-1th shuffling loop, respectively; $Tolerance_{obj}$ denotes the objective function convergence criterion value.

(b) Parameter convergence: When the algorithm cannot significantly change the parameter values over one or more iterations, the search should be stopped. The parameter convergence criterion is as follows:

$$\left|\frac{\rho_i^{(j)} - \rho_{i-1}^{(j)}}{\rho_{\max}^{(j)} - \rho_{\min}^{(j)}}\right| \leq Tolerance_{param}$$

where $\rho_i^{(j)}$ and $\rho_{i-1}^{(j)}$ denote the jth parameter values at the ith and i-1th shuffling loop, respectively; $\rho_{\max}^{(j)}$ and $\rho_{\min}^{(j)}$ denote the jth parameter's maximum and minimum values, respectively; $Tolerance_{param}$ denotes the parameter convergence criterion value.

(c) Maximum objective function evaluations: Maximum objective function evaluations is used as a backup to prevent wasting computing resources. If the number of objective function evaluations exceeds this criterion value, the search is stopped.

The CCE algorithm, based on the Nelder & Mead (1965) simplex downhill search scheme, is presented briefly as follows.

(I) Construct a subcomplex by randomly selecting q points from the complex according to a trapezoidal probability distribution. The probability distribution is specified such that the best point has the highest chance of being chosen to form the subcomplex, and the worst point has the least chance.

(II) Identify the worst point of the subcomplex and compute the centroid of the subcomplex without including the worst point.

(III) Attempt a reflection step by reflecting the worst point through the centroid. If the newly generated point is within the feasible space, go to Step IV; otherwise, randomly generate a point within the feasible space and go to Step VI.

(IV) If the newly generated point is better than the worst point, replace the worst point by the new point. Go to Step VII. Otherwise, go to Step V.

(V) Attempt a contraction step by computing a point halfway between the centroid and the worst point. If the contraction point is better than the worst point, replace the worst point by the contraction point and go to Step VII. Otherwise, go to Step VI.

(VI) Randomly generate a point within the feasible space. Replace the worst point by the randomly generated point.

(VII) Repeat Steps II–VI α times, where $\alpha \geq 1$ is the number of consecutive offspring generated by the same subcomplex.

(VIII) Repeat Steps I–VII β times, where $\beta \geq 1$ is the number of evolution steps taken by each complex before complexes are shuffled (Duan et al. 1994).

PARALLEL SCE-UA

The principle of the parallel SCE-UA method is stated as follows and its flow chart is demonstrated in Figure 1.

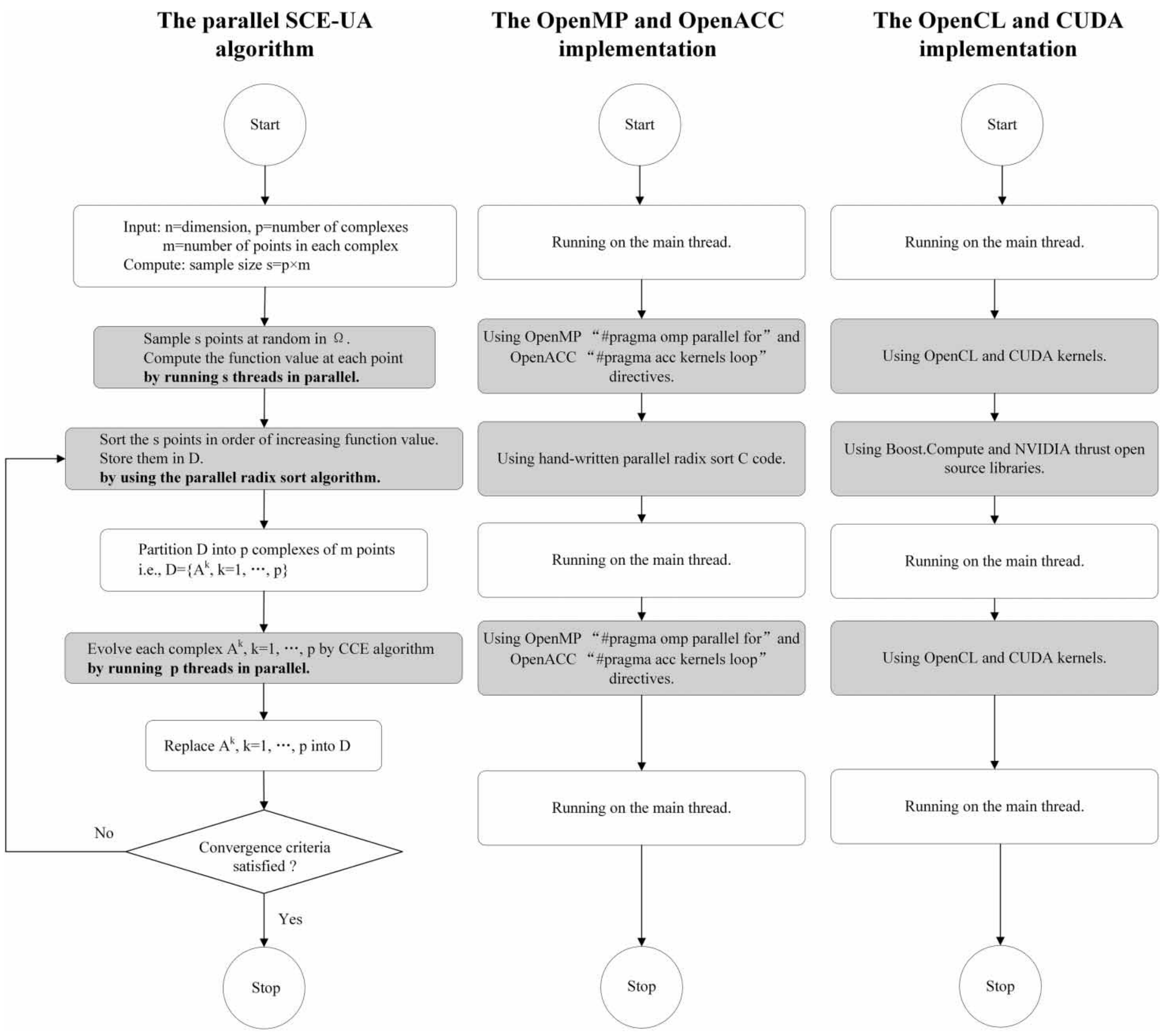

The parallel SCE-UA algorithm
The OpenMP and OpenACC implementation
The OpenCL and CUDA implementation
Start
Input: n=dimension, p=number of complexes m=number of points in each complex Compute: sample size s=p×m
Sample s points at random in Ω. Compute the function value at each point by running s threads in parallel.
Sort the s points in order of increasing function value. Store them in D. by using the parallel radix sort algorithm.
Partition D into p complexes of m points i.e., D={A^k, k=1, ⋯, p}
Evolve each complex A^k, k=1, ⋯, p by CCE algorithm by running p threads in parallel.
Replace A^k, k=1, ⋯, p into D
Convergence criteria satisfied ?
No
Yes
Stop
Running on the main thread.
Using OpenMP "#pragma omp parallel for" and OpenACC "#pragma acc kernels loop" directives.
Using hand-written parallel radix sort C code.
Using OpenCL and CUDA kernels.
Using Boost.Compute and NVIDIA thrust open source libraries.

Figure 1 | Flow chart of the parallel SCE-UA method.

(1) Generate sample in parallel. Create s parallel executed threads; in each thread sample a point randomly in the feasible parameter space and compute the criterion value at this point. In the absence of prior information, use a uniform probability distribution. For OpenMP and OpenACC implementations, this step is implemented through a for loop which is decorated by the '#pragma omp parallel for' OpenMP directive and the '#pragma acc kernels loop' OpenACC directive. For OpenCL and CUDA implementations, this step is implemented by using the OpenCL kernel and the CUDA kernel.

(2) Rank points. Sort the s points in parallel in order of increasing criterion value. For OpenMP and OpenACC implementations, this step is implemented by the hand-written parallel radix sort C/C++ code. For OpenCL implementation, this step is implemented by using the open source Boost.Compute library. For CUDA implementation, this step is implemented by using the open source NVIDIA thrust library.

(3) Partition into complexes. Partition the s points into p complexes, each containing m points. The complexes are partitioned such that the first complex contains every $p(k-1)+1$ ranked point, the second complex contains every $p(k-1)+2$ ranked point, and so on, where $k=1, 2, \ldots, m$. This step is run in serial.

(4) Evolve each complex. Create p parallel executed threads; in each thread evolve each complex according to the CCE

algorithm described above. For OpenMP and OpenACC implementations, this step is implemented through a for loop which is decorated by the '#pragma omp parallel for' OpenMP directive and the '#pragma acc kernels loop' OpenACC directive. For OpenCL and CUDA implementations, this step is implemented by using the OpenCL kernel and the CUDA kernel.

(5) Shuffle complexes. Combine the points in the evolved complexes into a single sample population (this is run in serial); sort the sample population in order of increasing criterion value (this is run in parallel, the same as Step 2); re-partition the sample population into p complexes according to the procedure specified in Step 3 (this is run in serial).

(6) Check convergence. If any of the pre-specified convergence criteria are satisfied, stop; otherwise, return to Step 4. This step is run in serial.

HARDWARE AND SOFTWARE USED IN THIS STUDY

In this study, the performance tests were carried out based on the Intel Xeon E5-2640v2 CPU and the NVIDIA Tesla K40c GPU. The CPU and GPU were hosted by a HP Z820 workstation. The hardware used in this study is listed in Table 1.

This study was based on the Microsoft Windows 7 platform. The source code was developed under the Microsoft Visual Studio 2010 Integrated Development Environment (IDE). The serial and OpenMP codes were developed by using the Microsoft Visual C++ 2010 compiler with OpenMP supported. The OpenCL and CUDA codes were developed by using the NVIDIA NVCC compiler, OpenCL 1.2, and CUDA 6.5. The OpenACC code was developed by using the PGI Accelerator workstation C/C++ 15.10 compiler with OpenACC supported (this compiler is run under the Cygwin by using the command shell).

RESULTS AND DISCUSSION

The serial and parallel SCE-UA methods were tested based on the Griewank benchmark problems. The performances were compared and the sensitivity was analyzed based on the testing results. As the calculation of the Griewank benchmark function is very fast and cannot mimic the heavy computational burden of the complex problem's objective function, such as hydrological model simulation, we added four kinds of arithmetic operations after the calculation of the Griewank function for each objective function evaluation. These operations include *nobj* additions, *nobj* subtractions, *nobj* multiplications, and *nobj* divisions of a dummy temporary variable. For the GPU version SCE-UA, the time consumed by the memory transfer is considered in the total execution time to ensure a fair comparison between the CPU and GPU versions.

The Griewank benchmark function is a relatively hard problem for optimization algorithms. It has lots of local minima which confuse the algorithm making the problem much harder. The Griewank function is as follows and the global minimum is 0 and is at the origin:

$$f(x) = \sum_{i=1}^{nopt} \frac{x_i^2}{d} - \prod_{i=1}^{nopt} \cos(x_i/\sqrt{i}) + 1$$

$$-600 \leq x_i \leq 600, \quad i = 1, \ldots, nopt, \quad d = 600$$

where x_i denotes the ith decision variable; *nopt* denotes the number of decision variables.

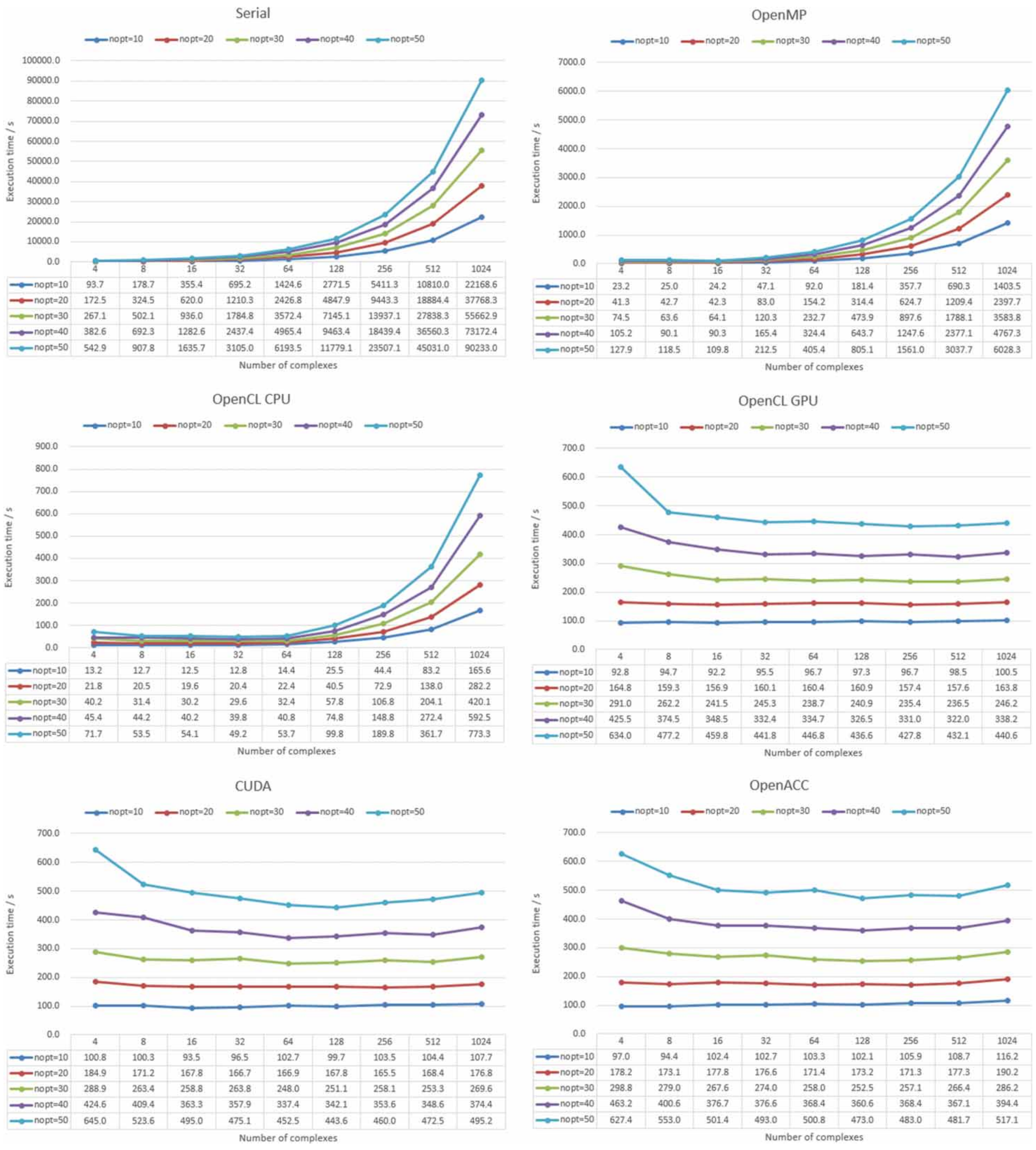
We carried out six comparisons which are shown below.

(1) Comparison of execution time of serial, OpenMP, OpenCL, CUDA, and OpenACC SCE-UA.

The execution time of the different SCE-UA implementations are listed in Figure 2. The execution time of the serial SCE-UA varied from 93.7 s to 90,233.0 s. The execution time of the OpenMP SCE-UA

Table 1 | Hardware used in this study

Computer name	Device name	Device description
HP Z820	CPU: Intel Xeon E5-2640v2	8 physical cores (16 logical cores), 2.0 GHz, 0.032TFLOPS single precision compute capability
	GPU1: NVIDIA Tesla K40c	2,880 cores, 875 MHz boost on, 12GB ECC memory, 4.29TFLOPS single precision compute capability
	GPU2: NVIDIA Quadro K6000	2,880 CUDA cores, 12GB ECC memory, used for display
	System memory: Samsung DDR3	32GB ECC memory
	Hard drive disk: West Digital	1TB SATA, 7,200 rpm



Serial

	4	8	16	32	64	128	256	512	1024
nopt=10	93.7	178.7	355.4	695.2	1424.6	2771.5	5411.3	10810.0	22168.6
nopt=20	172.5	324.5	620.0	1210.3	2426.8	4847.9	9443.3	18884.4	37768.3
nopt=30	267.1	502.1	936.0	1784.8	3572.4	7145.1	13937.1	27838.3	55662.9
nopt=40	382.6	692.3	1282.6	2437.4	4965.4	9463.4	18439.4	36560.3	73172.4
nopt=50	542.9	907.8	1635.7	3105.0	6193.5	11779.1	23507.1	45031.0	90233.0

OpenMP

	4	8	16	32	64	128	256	512	1024
nopt=10	23.2	25.0	24.2	47.1	92.0	181.4	357.7	690.3	1403.5
nopt=20	41.3	42.7	42.3	83.0	154.2	314.4	624.7	1209.4	2397.7
nopt=30	74.5	63.6	64.1	120.3	232.7	473.9	897.6	1788.1	3583.8
nopt=40	105.2	90.1	90.3	165.4	324.4	643.7	1247.6	2377.1	4767.3
nopt=50	127.9	118.5	109.8	212.5	405.4	805.1	1561.0	3037.7	6028.3

OpenCL CPU

	4	8	16	32	64	128	256	512	1024
nopt=10	13.2	12.7	12.5	12.8	14.4	25.5	44.4	83.2	165.6
nopt=20	21.8	20.5	19.6	20.4	22.4	40.5	72.9	138.0	282.2
nopt=30	40.2	31.4	30.2	29.6	32.4	57.8	106.8	204.1	420.1
nopt=40	45.4	44.2	40.2	39.8	40.8	74.8	148.8	272.4	592.5
nopt=50	71.7	53.5	54.1	49.2	53.7	99.8	189.8	361.7	773.3

OpenCL GPU

	4	8	16	32	64	128	256	512	1024
nopt=10	92.8	94.7	92.2	95.5	96.7	97.3	96.7	98.5	100.5
nopt=20	164.8	159.3	156.9	160.1	160.4	160.9	157.4	157.6	163.8
nopt=30	291.0	262.2	241.5	245.3	238.7	240.9	235.4	236.5	246.2
nopt=40	425.5	374.5	348.5	332.4	334.7	326.5	331.0	322.0	338.2
nopt=50	634.0	477.2	459.8	441.8	446.8	436.6	427.8	432.1	440.6

CUDA

	4	8	16	32	64	128	256	512	1024
nopt=10	100.8	100.3	93.5	96.5	102.7	99.7	103.5	104.4	107.7
nopt=20	184.9	171.2	167.8	166.7	166.9	167.8	165.5	168.4	176.8
nopt=30	288.9	263.4	258.8	263.8	248.0	251.1	258.1	253.3	269.6
nopt=40	424.6	409.4	363.3	357.9	337.4	342.1	353.6	348.6	374.4
nopt=50	645.0	523.6	495.0	475.1	452.5	443.6	460.0	472.5	495.2

OpenACC

	4	8	16	32	64	128	256	512	1024
nopt=10	97.0	94.4	102.4	102.7	103.3	102.1	105.9	108.7	116.2
nopt=20	178.2	173.1	177.8	176.6	171.4	173.2	171.3	177.3	190.2
nopt=30	298.8	279.0	267.6	274.0	258.0	252.5	257.1	266.4	286.2
nopt=40	463.2	400.6	376.7	376.6	368.4	360.6	368.4	367.1	394.4
nopt=50	627.4	553.0	501.4	493.0	500.8	473.0	483.0	481.7	517.1

Figure 2 | Execution time of serial, OpenMP, OpenCL, CUDA, and OpenACC SCE-UA.

varied from 23.2 s to 6,028.3 s. The execution time of the OpenCL CPU SCE-UA varied from 12.5 s to 773.3 s. The execution time of the OpenCL GPU SCE-UA varied from 92.2 s to 634.0 s. The execution time of the CUDA SCE-UA varied from 93.5 s to 645.0 s. The execution time of the OpenACC SCE-UA varied from 94.4 s to 627.4 s. We can see that the serial SCE-UA runs very slowly. The OpenMP version runs faster than the serial version. The OpenCL CPU version runs faster than the OpenMP version. The

GPU versions (including the OpenCL GPU, CUDA, and OpenACC versions) run very fast when the number of complexes became very large and run slower than the serial version when the number of complexes is small. When the number of complexes increased, the CPU versions (including the serial, OpenMP and OpenCL CPU versions) consumed more and more time. However, the execution time of the GPU versions did not change significantly.

(2) Comparison of speedup ratio of OpenMP, OpenCL, CUDA, and OpenACC SCE-UA versus serial SCE-UA.

The speedup ratio of different SCE-UA implementations are listed in Figure 3. The speedup ratio of the OpenMP SCE-UA varied from 3.59× to 15.80×. The speedup ratio of the OpenCL CPU SCE-UA varied from 6.65× to 136.84×. The speedup ratio of the OpenCL GPU SCE-UA varied from 0.86× to 230.60×. The speedup ratio of the CUDA SCE-UA varied from 0.84× to 213.59×. The speedup ratio of the OpenACC SCE-UA varied from 0.83× to 198.61×. We can see that with the increasing number of complexes (i.e., the number of parallel executed threads), the parallel SCE-UA runs faster than the serial version. For a smaller number of complexes, the OpenMP and OpenCL CPU versions run faster. For large numbers of complexes, the GPU versions run faster. The OpenCL GPU SCE-UA run faster than the CUDA SCE-UA. The CUDA SCE-UA run faster than the OpenACC SCE-UA.

(3) Analysis of the impact of objective function computational load.

Here we fixed *nopt* to 20 (because SCE-UA are mostly applied in the field of hydrological model parameter calibration and most hydrological models have less than 20 parameters) and changed the *p* (i.e., the number of complexes) and *nobj* (reflecting the objective function computational load) to test the impact of objective function computational load on the speedup ratio. We can see in Figure 4 that with the increasing of the *nobj*, the speedup ratio increased. This means that the speedup ratio can be larger with increasing objective function computational load. With the increasing of the *p*, the GPU versions run increasingly faster. This fact indicates that the GPU versions run very fast for large *p* and the CPU versions run very fast for small *p*.

(4) Analysis of impact of error correction code (ECC) setting for GPU device.

The Tesla GPU used in this study supported ECC. We tested the impact of ECC setting to the execution time. The execution time ratio of ECC on versus ECC off is demonstrated in Figure 5. For OpenCL GPU SCE-UA, the number of ratio large than 1, smaller than 1, and equal to 1 was 17, 11, and 17 respectively, whilst for CUDA SCE-UA, it was 14, 17, and 14, respectively. For OpenACC SCE-UA, the number of ratio large than 1, smaller than 1, and equal to 1 was 36, 9, and 0, respectively. The results indicate that from the point of view of average and overall performance, the OpenCL GPU version runs slower when ECC is on, the CUDA version runs faster when ECC is on, and the OpenACC version runs slower when ECC is on. The OpenACC version is most sensitive to the ECC on setting.

(5) Comparison of source code complexity.

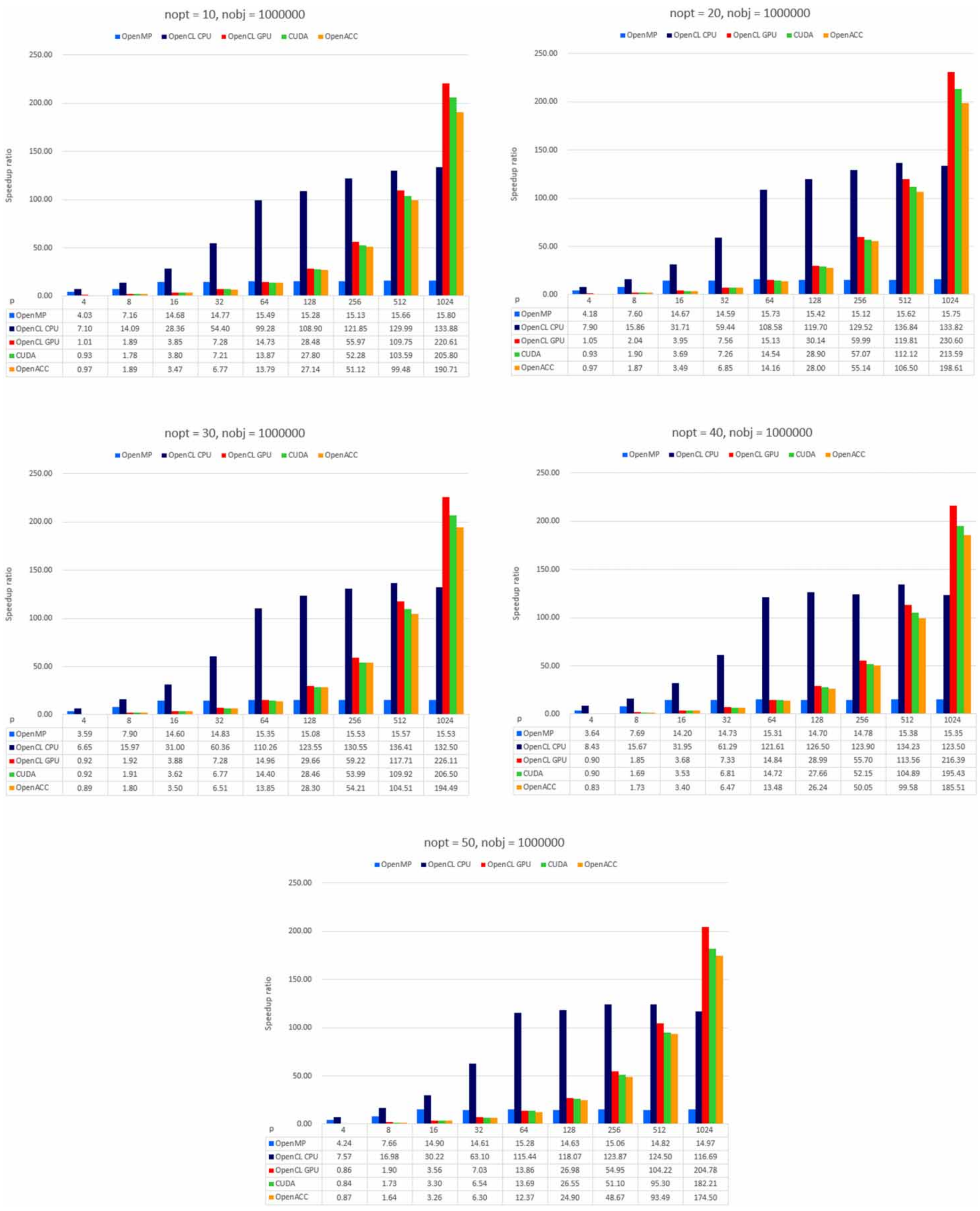
The source code length was analyzed here. The lengths of the serial, OpenMP, OpenCL CPU, OpenCL GPU, CUDA, and OpenACC implementations were 941, 941, 1,972, 1,972, 1,402, and 1,005, respectively. Considering the acceleration results mentioned above, although the OpenCL versions obtained the highest overall speedup ratio, they were the most complex implementation. The CUDA version is the moderate complex implementation. The complexity of the OpenMP and OpenACC versions is almost same as the original serial version.

(6) Comparison of optimization accuracy.

We tested all the optimizations and found that all optimizations converged to the theoretical global optimum (i.e., the origin). This result indicated that the optimization accuracy of the parallel SCE-UA is satisfactory.

CONCLUSIONS

This paper aimed at the parallelization and acceleration of the SCE-UA method based on powerful heterogeneous computing technology. The parallel SCE-UA was implemented on an Intel Xeon multi-core CPU (by using OpenMP and OpenCL) and a NVIDIA Tesla many-core GPU (by using OpenCL, CUDA, and OpenACC). The serial and parallel SCE-UA methods were tested based on the Griewank benchmark function. Comparison results indicate the parallel SCE-UA significantly improved the computation efficiency compared to the original serial version. The OpenCL implementation obtained the best overall acceleration results, however it is the most complex source code. The



p	4	8	16	32	64	128	256	512	1024
OpenMP	4.03	7.16	14.68	14.77	15.49	15.28	15.13	15.66	15.80
OpenCL CPU	7.10	14.09	28.36	54.40	99.28	108.90	121.85	129.99	133.88
OpenCL GPU	1.01	1.89	3.85	7.28	14.73	28.48	55.97	109.75	220.61
CUDA	0.93	1.78	3.80	7.21	13.87	27.80	52.28	103.59	205.80
OpenACC	0.97	1.89	3.47	6.77	13.79	27.14	51.12	99.48	190.71

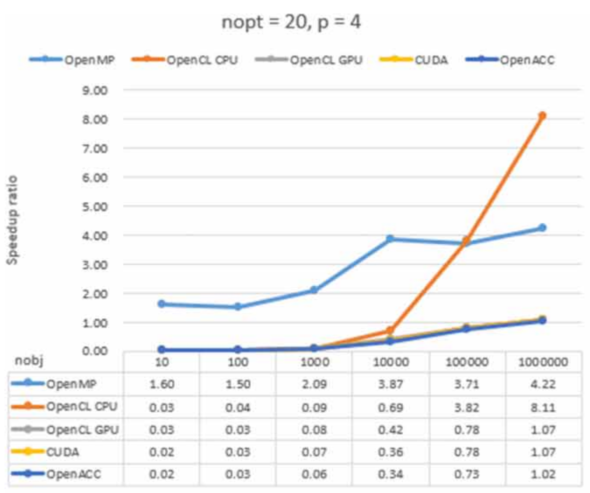
p	4	8	16	32	64	128	256	512	1024
OpenMP	4.18	7.60	14.67	14.59	15.73	15.42	15.12	15.62	15.75
OpenCL CPU	7.90	15.86	31.71	59.44	108.58	119.70	129.52	136.84	133.82
OpenCL GPU	1.05	2.04	3.95	7.56	15.13	30.14	59.99	119.81	230.60
CUDA	0.93	1.90	3.69	7.26	14.54	28.90	57.07	112.12	213.59
OpenACC	0.97	1.87	3.49	6.85	14.16	28.00	55.14	106.50	198.61

p	4	8	16	32	64	128	256	512	1024
OpenMP	3.59	7.90	14.60	14.83	15.35	15.08	15.53	15.57	15.53
OpenCL CPU	6.65	15.97	31.00	60.36	110.26	123.55	130.55	136.41	132.50
OpenCL GPU	0.92	1.92	3.88	7.28	14.96	29.66	59.22	117.71	226.11
CUDA	0.92	1.91	3.62	6.77	14.40	28.46	53.99	109.92	206.50
OpenACC	0.89	1.80	3.50	6.51	13.85	28.30	54.21	104.51	194.49

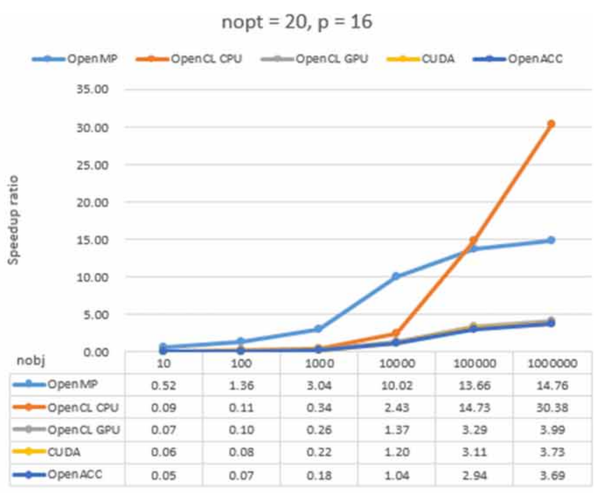
p	4	8	16	32	64	128	256	512	1024
OpenMP	3.64	7.69	14.20	14.73	15.31	14.70	14.78	15.38	15.35
OpenCL CPU	8.43	15.67	31.95	61.29	121.61	126.50	123.90	134.23	123.50
OpenCL GPU	0.90	1.85	3.68	7.33	14.84	28.99	55.70	113.56	216.39
CUDA	0.90	1.69	3.53	6.81	14.72	27.66	52.15	104.89	195.43
OpenACC	0.83	1.73	3.40	6.47	13.48	26.24	50.05	99.58	185.51

p	4	8	16	32	64	128	256	512	1024
OpenMP	4.24	7.66	14.90	14.61	15.28	14.63	15.06	14.82	14.97
OpenCL CPU	7.57	16.98	30.22	63.10	115.44	118.07	123.87	124.50	116.69
OpenCL GPU	0.86	1.90	3.56	7.03	13.86	26.98	54.95	104.22	204.78
CUDA	0.84	1.73	3.30	6.54	13.69	26.55	51.10	95.30	182.21
OpenACC	0.87	1.64	3.26	6.30	12.37	24.90	48.67	93.49	174.50

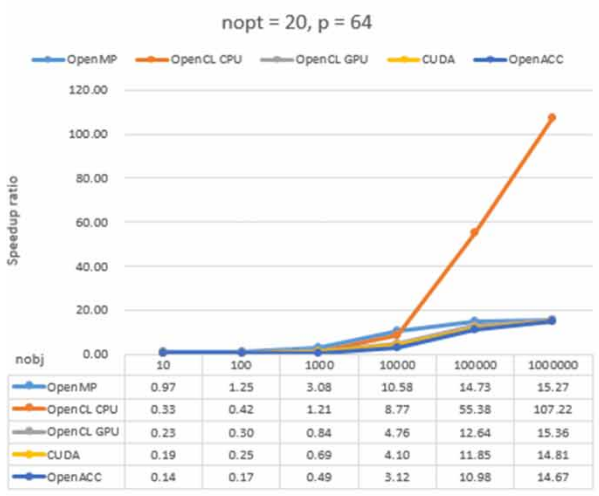
Figure 3 | Speedup ratio of OpenMP, OpenCL, CUDA, and OpenACC SCE-UA versus serial SCE-UA.



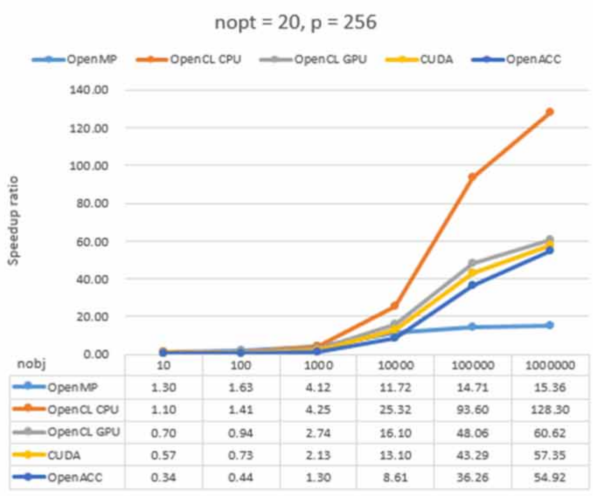
nobj	10	100	1000	10000	100000	1000000
OpenMP	1.60	1.50	2.09	3.87	3.71	4.22
OpenCL CPU	0.03	0.04	0.09	0.69	3.82	8.11
OpenCL GPU	0.03	0.03	0.08	0.42	0.78	1.07
CUDA	0.02	0.03	0.07	0.36	0.78	1.07
OpenACC	0.02	0.03	0.06	0.34	0.73	1.02



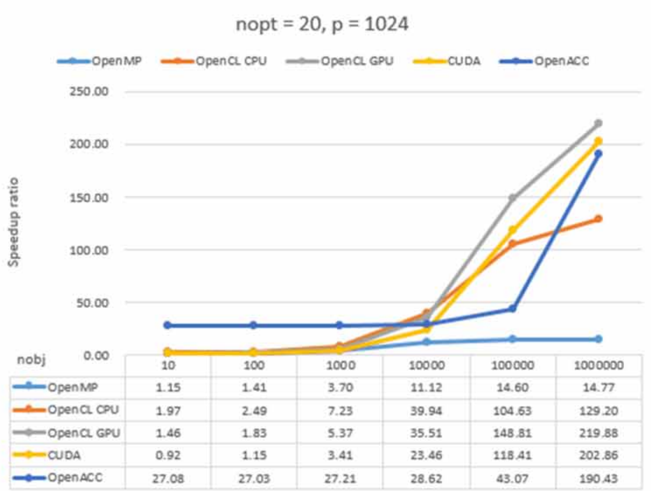
nobj	10	100	1000	10000	100000	1000000
OpenMP	0.52	1.36	3.04	10.02	13.66	14.76
OpenCL CPU	0.09	0.11	0.34	2.43	14.73	30.38
OpenCL GPU	0.07	0.10	0.26	1.37	3.29	3.99
CUDA	0.06	0.08	0.22	1.20	3.11	3.73
OpenACC	0.05	0.07	0.18	1.04	2.94	3.69



nobj	10	100	1000	10000	100000	1000000
OpenMP	0.97	1.25	3.08	10.58	14.73	15.27
OpenCL CPU	0.33	0.42	1.21	8.77	55.38	107.22
OpenCL GPU	0.23	0.30	0.84	4.76	12.64	15.36
CUDA	0.19	0.25	0.69	4.10	11.85	14.81
OpenACC	0.14	0.17	0.49	3.12	10.98	14.67

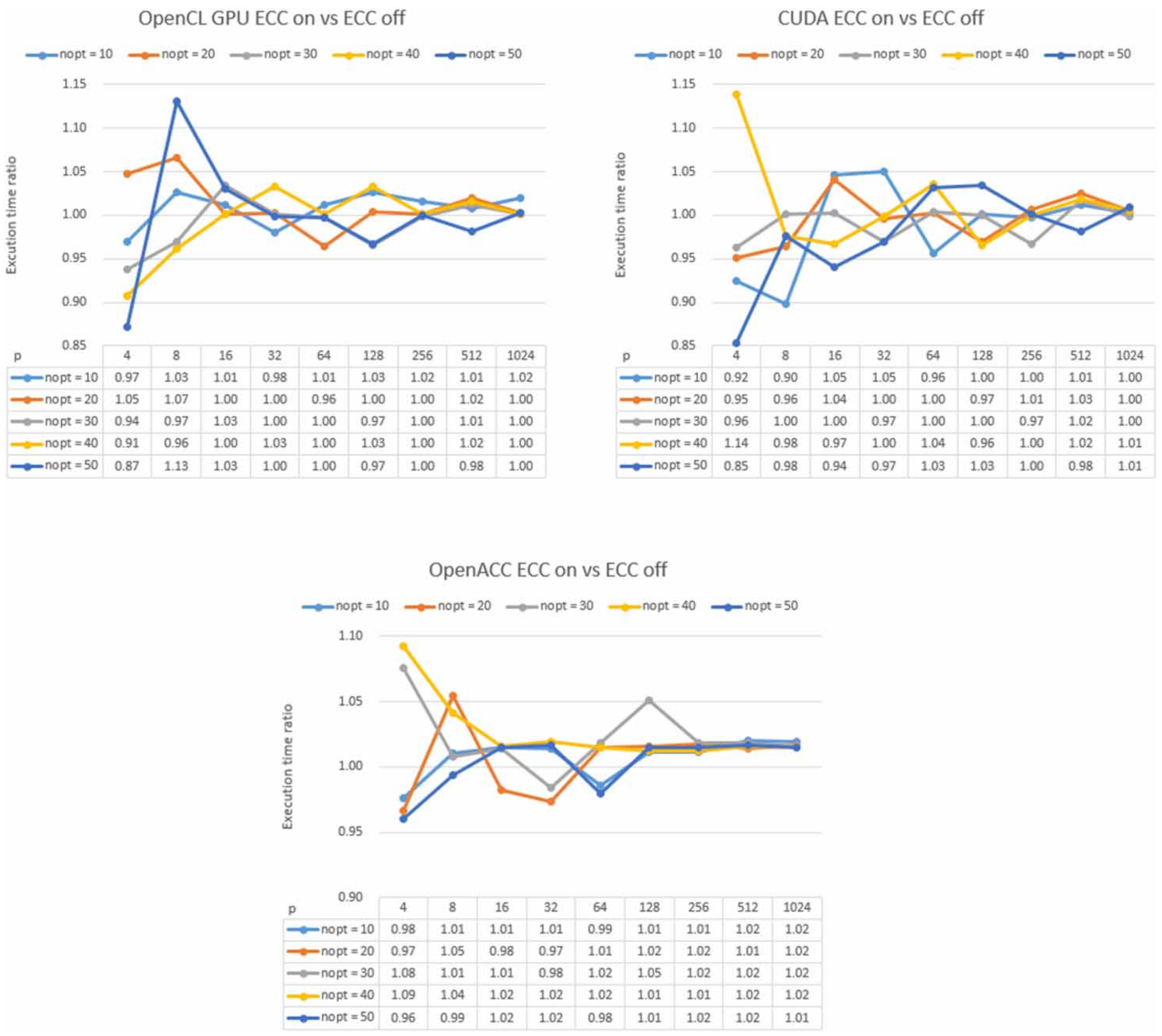


nobj	10	100	1000	10000	100000	1000000
OpenMP	1.30	1.63	4.12	11.72	14.71	15.36
OpenCL CPU	1.10	1.41	4.25	25.32	93.60	128.30
OpenCL GPU	0.70	0.94	2.74	16.10	48.06	60.62
CUDA	0.57	0.73	2.13	13.10	43.29	57.35
OpenACC	0.34	0.44	1.30	8.61	36.26	54.92



nobj	10	100	1000	10000	100000	1000000
OpenMP	1.15	1.41	3.70	11.12	14.60	14.77
OpenCL CPU	1.97	2.49	7.23	39.94	104.63	129.20
OpenCL GPU	1.46	1.83	5.37	35.51	148.81	219.88
CUDA	0.92	1.15	3.41	23.46	118.41	202.86
OpenACC	27.08	27.03	27.21	28.62	43.07	190.43

Figure 4 | Speedup ratio comparison for different objective function computational load settings.



OpenCL GPU ECC on vs ECC off

p	4	8	16	32	64	128	256	512	1024
nopt = 10	0.97	1.03	1.01	0.98	1.01	1.03	1.02	1.01	1.02
nopt = 20	1.05	1.07	1.00	1.00	0.96	1.00	1.00	1.02	1.00
nopt = 30	0.94	0.97	1.03	1.00	1.00	0.97	1.00	1.01	1.00
nopt = 40	0.91	0.96	1.00	1.03	1.00	1.03	1.00	1.02	1.00
nopt = 50	0.87	1.13	1.03	1.00	1.00	0.97	1.00	0.98	1.00

CUDA ECC on vs ECC off

p	4	8	16	32	64	128	256	512	1024
nopt = 10	0.92	0.90	1.05	1.05	0.96	1.00	1.00	1.01	1.00
nopt = 20	0.95	0.96	1.04	1.00	1.00	0.97	1.01	1.03	1.00
nopt = 30	0.96	1.00	1.00	0.97	1.00	1.00	0.97	1.02	1.00
nopt = 40	1.14	0.98	0.97	1.00	1.04	0.96	1.00	1.02	1.01
nopt = 50	0.85	0.98	0.94	0.97	1.03	1.03	1.00	0.98	1.01

OpenACC ECC on vs ECC off

p	4	8	16	32	64	128	256	512	1024
nopt = 10	0.98	1.01	1.01	1.01	0.99	1.01	1.01	1.02	1.02
nopt = 20	0.97	1.05	0.98	0.97	1.01	1.02	1.02	1.01	1.02
nopt = 30	1.08	1.01	1.01	0.98	1.02	1.05	1.02	1.02	1.02
nopt = 40	1.09	1.04	1.02	1.02	1.02	1.01	1.01	1.02	1.02
nopt = 50	0.96	0.99	1.02	1.02	0.98	1.01	1.02	1.02	1.01

Figure 5 | Comparison of execution time ratio of ECC on versus ECC off.

parallel SCE-UA has bright prospects to be applied in real-world applications.

ACKNOWLEDGEMENTS

This research was funded by the China Postdoctoral Science Foundation on Grant (Grant No. 2016M600096); Research on Key Technologies of Real Time Dynamic Control of Water Level in Flood Season of Cascade Reservoirs (KY1734); 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, SLXMB200902); Construction project of Shaanxi province medium and small river hydrological monitoring and forecast system – construction of Guanzhong and north of Shaanxi flood forecast scheme (JZ0205A112015); National Key Research and Development Plan (2016YFC0803107); Major International (Regional) Joint Research Project – China's Water and Food Security under Extreme Climate Change Impact: Risk Assessment and Resilience (G0305, 7141101024), International Project (71461010701); Study of distributed flood risk forecast model and technology based on multi-source data integration and hydro meteorological coupling system (2013CB036406); and China National Flash Flood Disaster Prevention and Control Project (126301001000150068). We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research. The authors wish to acknowledge Dr Mark Amo-Boateng for his help in developing the CUDA-based SCE-UA method. Guangyuan Kan and Ke Liang contributed equally to this work.

REFERENCES

Dong, J., Zheng, C., Kan, G., Wen, J., Zhao, M. & Yu, J. 2015 Applying the ensemble artificial neural network-based hybrid data-driven model to daily total load forecasting. *Neural Computing & Applications* **26** (3), 603–611.

Duan, Q. 1991 *A Global Optimization Strategy for Efficient and Effective Calibration of Hydrologic Models*. PhD Thesis, Department of Hydrology and Water Resources, University of Arizona, Tucson, AZ, USA.

Duan, Q., Sorooshian, S. & Gupta, V. 1992 Effective and efficient global optimization for conceptual rainfall-runoff models. *Water Resources Research* **28** (4), 1015–1031.

Duan, Q., Gupta, V. & Sorooshian, S. 1993 A shuffled complex evolution approach for effective and efficient global minimization. *Journal of Optimization Theory and Applications* **76** (3), 501–521.

Duan, Q., Sorooshian, S. & Gupta, V. 1994 Optimal use of the SCE-UA global optimization method for calibrating watershed models. *Journal of Hydrology* **158** (3–4), 265–284.

Gong, W., Duan, Q., Li, J., Wang, C., Di, Z., Dai, Y., Ye, A. & Miao, C. 2015 Multi-objective parameter optimization of common land model using adaptive surrogate modeling. *Hydrology and Earth System Sciences* **19**, 2409–2425.

Gong, W., Duan, Q., Li, J., Wang, C., Di, Z., Ye, A., Miao, C. & Dai, Y. 2016 Multiobjective adaptive surrogate modeling-based optimization for parameter estimation of large, complex geophysical models. *Water Resources Research* **52**, 1984–2008.

Holland, J. H. 1975 *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, USA.

Kan, G., Yao, C., Li, Q., Li, Z., Yu, Z., Liu, Z., Ding, L., He, X. & Liang, K. 2015a Improving event-based rainfall-runoff simulation using an ensemble artificial neural network based hybrid data-driven model. *Stochastic Environmental Research and Risk Assessment* **29**, 1345–1370.

Kan, G., Li, J., Zhang, X., Ding, L., He, X., Liang, K., Jiang, X., Ren, M., Li, H., Wang, F., Zhang, Z. & Hu, Y. 2015b A new hybrid data-driven model for event-based rainfall-runoff simulation. *Neural Computing & Applications* doi:10.1007/s00521-016-2200-4.

Kan, G., He, X., Li, J., Ding, L., Zhang, D., Lei, T., Hong, Y., Liang, K., Zuo, D., Bao, Z. & Zhang, M. 2016a A novel hybrid data-driven model for multi-input single-output system simulation. *Neural Computing & Applications* doi:10.1007/s00521-016-2534-y.

Kan, G., Liang, K., Li, J., Ding, L., He, X., Hu, Y. & Amo-Boateng, M. 2016b Accelerating the SCE-UA global optimization method based on multi-core CPU and many-core GPU. *Advances in Meteorology* http://dx.doi.org/10.1155/2016/8483728.

Kan, G., Lei, T., Liang, K., Li, J., Ding, L., He, X., Yu, H., Zhang, D., Zuo, D., Bao, Z., Amo-Boateng, A., Hu, Y. & Zhang, M. 2016c A multi-core CPU and many-core GPU based fast parallel shuffled complex evolution global optimization approach. *IEEE Transactions on Parallel and Distributed Systems* doi:10.1109/TPDS.2016.2575822.

Kan, G., Zhang, M., Liang, K., Wang, H., Jiang, Y., Li, J., Ding, L., He, X., Hong, Y., Zuo, D., Bao, Z. & Li, C. 2016d Improving water quantity simulation and forecasting to solve the energy-water-food nexus issue by using heterogeneous computing accelerated global optimization method. *Applied Energy* http://dx.doi.org/10.1016/j.apenergy.2016.08.017.

Kan, G., He, X., Ding, L., Li, J., Lei, T., Liang, K. & Hong, Y. 2016e An improved hybrid data-driven model and its application in daily rainfall-runoff simulation. *IOP Conference Series: Earth and Environmental Science* **46** (1), 012029 (6th Digital Earth Summit), doi:10.1088/1755-1315/46/1/012029.

Li, Z., Kan, G., Yao, C., Liu, Z., Li, Q. & Yu, S. 2014 An improved neural network model and its application in hydrological simulation. *Journal of Hydrologic Engineering* **19** (10), 04014019-1–04014019-17.

Li, C., Cheng, X., Li, N., Du, X., Yu, Q. & Kan, G. 2016 A framework for flood risk analysis and benefit assessment of flood control measures in urban areas. *International Journal of Environmental Research and Public Health* **13**, 787. doi:10.3390/ijerph13080787.

Muttil, N., Liong, S. Y. & Nesterov, O. 2007 A parallel shuffled complex evolution model calibrating algorithm to reduce computational time. In: *MODSIM 2007 International Congress on Modelling and Simulation*. 10–13 December 2007, Christchurch, New Zealand.

Nelder, J. A. & Mead, R. 1965 A simplex method for function minimization. *The Computer Journal* **7**, 308–313.

NVIDIA Corporation 2015 *CUDA C Programming Guide*. Santa Clara, CA, USA.

Sharma, V., Swayne, D. A., Lam, D. & Schertzer, W. 2006 Parallel shuffled complex evolution algorithm for calibration of hydrological models. In: *Proceedings of the 20th International Symposium on High-Performance Computing in an Advanced Collaborative Environment (HPCS'06)*, 14–17 May 2006, St Johns, Newfoundland, Canada, pp. 30–33.

Sorooshian, S., Duan, Q. & Gupta, V. K. 1993 Calibration of conceptual rainfall-runoff models using global optimization: application to the Sacramento soil moisture accounting model. *Water Resources Research* **29** (4), 1185–1194.

Wang, C., Duan, Q., Gong, W., Ye, A., Di, Z. & Miao, C. 2014 An evaluation of adaptive surrogate modeling based optimization with two benchmark problems. *Environmental Modelling & Software* **60**, 167–179.

Zhang, J., Wang, X., Liu, P., Lei, X., Li, Z., Gong, W., Duan, Q. & Wang, H. 2017 Assessing the weighted multi-objective adaptive surrogate model optimization to derive large-scale reservoir operating rules with sensitivity analysis. *Journal of Hydrology* **544**, 613–627.

Zuo, D., Cai, S., Xu, Z., Li, F., Sun, W., Yang, X., Kan, G. & Liu, P. 2016 Spatiotemporal patterns of drought at various time scales in Shandong Province of Eastern China. *Theoretical and Applied Climatology* doi:10.1007/s00704-016-1969-5.

First received 20 September 2016; accepted in revised form 15 May 2017. Available online 6 June 2017