Hindawi
Complexity
Volume 2020, Article ID 6589658, 17 pages
https://doi.org/10.1155/2020/6589658

WILEY | Hindawi

Research Article

Parallelizing Comprehensive Learning Particle Swarm Optimization by Open Computing Language on an Integrated Graphical Processing Unit

Xiang Yu,[1] **Yu Qiao**,[2] **Qingpeng Li**,[3] **Gang Xu**,[4] **Chuanxiong Kang**,[1] **Claudio Estevez**,[5] **Chengzhi Deng**,[1] **and Shengqian Wang**[1]

[1]*Provincial Key Laboratory for Water Information Cooperative Sensing and Intelligent Processing, Nanchang Institute of Technology, Nanchang, Jiangxi 330099, China*
[2]*School of Mathematics and Information Science, Shaanxi Normal University, Xi'an, Shaanxi 710119, China*
[3]*State Grid Nanchang Electric Power Supply Company, Nanchang, Jiangxi 330069, China*
[4]*Department of Mathematics, Nanchang University, Nanchang, Jiangxi 330031, China*
[5]*Department of Electrical Engineering, University of Chile, Santiago 8370451, Chile*

Correspondence should be addressed to Xiang Yu; xiang.yu@nit.edu.cn

Received 22 April 2020; Revised 27 June 2020; Accepted 7 July 2020; Published 31 July 2020

Academic Editor: Eric Campos

Comprehensive learning particle swarm optimization (CLPSO) is a powerful metaheuristic for global optimization. This paper studies parallelizing CLPSO by open computing language (OpenCL) on the integrated Intel HD Graphics 520 (IHDG520) graphical processing unit (GPU) with a low clock rate. We implement a coarse-grained all-GPU model that maps each particle to a separate work item. Two enhancement strategies, namely, generating and transferring random numbers from the central processor to the GPU as well as reducing the number of instructions in the kernel, are proposed to shorten the model's execution time. This paper further investigates parallelizing deterministic optimization for implicit stochastic optimization of China's Xiaowan Reservoir. The deterministic optimization is performed on an ensemble of 62 years' historical inflow records with monthly time steps, is solved by CLPSO, and is parallelized by a coarse-grained multipopulation model extended from the all-GPU model. The multipopulation model involves a large number of work items. Because of the capacity limit for a buffer transferring data from the central processor to the GPU and the size of the global memory region, the random number generation strategy is modified by generating a small number of random numbers that can be flexibly exploited by the large number of work items. Experiments conducted on various benchmark functions and the case study demonstrate that our proposed all-GPU and multipopulation parallelization models are appropriate; and the multipopulation model achieves the consumption of significantly less execution time than the corresponding sequential model.

1. Introduction

A graphical processing unit (GPU) is a processor specially designed to rapidly manipulate the creation of images in a frame buffer intended for output to a display device. By providing functionalities such as texture mapping, rendering, shading, anti-aliasing, color space, and video decoding, a GPU is an indispensable aid to a central processing unit (CPU) to manage and boost the performance of graphics. A CPU consists of only a few processing elements optimized for sequential processing, whereas a GPU consists of a large number of compute units, with each compute unit in turn containing many processing elements, thereby constituting a massively parallel architecture for handling multiple computing tasks simultaneously. People have recently studied leveraging the massively parallel architectures of GPUs for accelerating nongraphical general-purpose computing in a wide range of areas [1–14]. GPU-based parallel computing is implemented by a host program and kernel(s). The host program runs on the CPU and can launch a kernel on a

connected GPU. A kernel is a function executed in parallel on the processing elements within the GPU. The parallel threads of the kernel can be organized into groups, with each group of concurrent threads executed on the same compute unit. Data are transferred between the CPU and the GPU.

Metaheuristics are naturally suitable to be parallelized on GPUs. A metaheuristic is essentially a set of nature-inspired intelligent search strategies and is promising for single-objective global optimization [15]. A metaheuristic usually finds the optimum using a population of individuals, with each individual representing a candidate solution. All the individuals are initialized randomly and evolve iteratively and randomly in the search space to gradually locate the optimum. Each individual is thus associated with a fitness indicating the individual's search performance. The final solution obtained is the individual that exhibits the best fitness among the population when the evolution ends. Compared with traditional optimization methods such as linear programming, nonlinear programming, dynamic programming, and optimal control, metaheuristics do not require the objective and constraints of the optimization problem to be continuous, differentiable, linear, or convex. In addition, metaheuristics can be directly linked with simulation models without requiring simplifying any assumption in the models. According to Tan and Ding [16], GPU-based parallelization implementations of metaheuristics are classified into four categories, namely, naive model, multiphase model, all-GPU model, and multipopulation model. The naive model offloads fitness calculation onto the GPU side. The naive model can be coarse-grained or fine-grained. For the coarse-grained naive model, the fitness of each individual is calculated by a separate thread. If the fitness is an aggregation of a number of partial calculations that can be run in parallel, then the fined-grained naive model can be used to further accelerate fitness calculation. Besides fitness calculation, the multiphase model also parallelizes other phases of a metaheuristic to leverage the GPU's computing power. Multiple kernels are used to parallelize different phases, and different kernels can take different parallelization granularities to best fit the corresponding tasks. The multiphase model might incur the overhead of frequent kernel launching. The all-GPU model combines multiple kernels into one kernel. Instead of relying on a single population, multiple populations are employed in the multipopulation model, with each population evolving separately with different thread(s).

Particle swarm optimization (PSO) is a class of metaheuristics simulating the food-searching behavior of a bird flock [17, 18]. In PSO, population is termed as swarm, and an individual is termed as the particle. All the particles "fly" in the search space. Each particle is accordingly additionally associated with a position and velocity. Each particle updates the flight trajectory iteratively and randomly and tries to gradually move towards the optimum. Many PSO variants have been proposed in the literature since 1995 [17, 18], and some variants have been parallelized on GPUs. Open computing language (OpenCL) [19] and compute unified device architecture (CUDA) [20] are two commonly used industry standards that facilitate parallel computing on GPUs. Based on the C/C++ programming language, OpenCL and CUDA both provide a series of runtime application programming interfaces (APIs). OpenCL enables programming across heterogeneous types of processors including not only GPU but also digital signal processor, and field programming gate array, whereas CUDA only supports NVIDIA proprietary GPUs. Global PSO (GPSO) is the earliest PSO variant [17]. In GPSO, the flight of each particle is guided by the particle's historical best search experience (i.e., personal best position) and also the historical best search experience out of the entire swarm (i.e., global best position). GPSO was parallelized on GPUs by CUDA following the multiphase model in [21], the all-GPU model in [1–3, 22], and the multipopulation model in [4, 5]. GPSO is liable to get stuck in premature convergence, and local PSO (LPSO) [23] is an improved variant based on GPSO. A static topology is constructed in LPSO. For a particle, its neighborhood contains itself and the particles that it directly connects to in the topology. Instead of referring to the global best position, the historical best search experience of the neighborhood (i.e., local best position) is used for updating particle's velocity. An all-GPU parallelization implementation of LPSO by CUDA was proposed in [6]. In contrast with LPSO, standard PSO (SPSO) [24] maintains a dynamic topology. An all-GPU parallelization model of SPSO implemented by CUDA and that by OpenCL were, respectively, introduced in [25, 26]. Comprehensive learning PSO (CLPSO) [27–29] differs from GPSO, LPSO, and SPSO in that CLPSO encourages each particle to learn from different exemplars on different dimensions during the flight trajectory update, whereas GPSO, LPSO, and PSO enforce the same pair of personal best position and global/local best position for updating particle's velocity on all the dimensions. CLPSO is good at preserving the particles' diversity and exhibits excellent optimization performance. Papadakis and Bakrtzis [7] investigated developing an all-GPU model of CLPSO by OpenCL. Kilic et al. [8], Ouyang et al. [9], Souza et al. [10], Sharma et al. [11, 12], and Rabinovich [30] studied parallelizing other PSO variants on GPUs.

In this paper, we study parallelizing CLPSO on a platform with the Intel Core i7-6500U CPU, the third-generation double data rate (DDR3) main memory, and the Intel HD Graphics 520 (IHDG520) GPU by OpenCL. IHDG520 is an integrated GPU; it can be found in various ultralow-voltage mobile CPUs and is suited for laptop (particularly, ultrabook) computers. The IHDG520 GPU lacks dedicated graphics memory and has to access the main memory. The sequentialization implementation of CLPSO was evaluated with a swarm of 40 particles in [27–29]. We propose a coarse-grained all-GPU model with the kernel being concurrently executed by 40 threads. Two enhancement strategies, i.e., generating and transferring random numbers from the CPU to the IHDG520 GPU as well as reducing the number of instructions in the kernel, are introduced into the model for the purpose of shortening the execution time. We further investigate parallelizing deterministic optimization for implicit stochastic optimization (ISO) of China's Xiaowan Reservoir. The deterministic optimization is performed on an ensemble of 62 years' historical inflow records with

monthly time steps and is solved by CLPSO. The deterministic optimization is parallelized by a coarse-grained multipopulation model extended from the all-GPU model, using each swarm of particles to address the optimal operation in a separate year. The multipopulation model involves a large number of threads. Because of the limit on data transfer capacity between the CPU and the IHDG520 GPU, we modify the random number generation strategy by just generating a small number of random numbers that can be flexibly exploited by the large number of threads without hurting randomness.

A reservoir is a hydraulic structure that impounds water and uses water to serve various purposes such as hydropower generation, flood control, navigation, sediment control, and water provisioning for agricultural, domestic, and industrial demands. A reservoir system consists of one or more cascaded reservoirs constructed within the same river basin. The optimal operation of a reservoir system is to schedule outflows of the reservoir (s) over a series of consecutive time steps in order to optimize a specific objective, trying to fulfill the multipurpose development of the system. The optimal operation of a reservoir system is complex because the optimization problem has to take into account inflow imprecision and uncertainties, the dynamic multistage nature of decision-making, and different physical and operational constraints [31–36]. The optimal operation of a reservoir system is either deterministic or stochastic. Deterministic optimization assumes that inflows into the reservoirs (s) over all the time steps are available. However, in practice, only limited inflow forecasting information can be obtained. Alternatives to avoid perfect inflow knowledge during the entire planning horizon include ISO and explicit stochastic optimization (ESO) [35, 37]. ISO is also referred to as Monte Carlo optimization. ISO optimizes over a long continuous series of historically recorded or synthetically generated inflows, or many shorter equally likely sequences. Accordingly, stochasticity of the inflows is implicitly addressed, and deterministic optimization can be applied on the ensemble of inflows. Operation rules for the outflows of each reservoir over all the time steps conditioned on information, e.g., the reservoir's present storage volume (or forebay elevation), previous inflows, and limited forecasted inflows, are then abstracted from the deterministic optimization results using a fitting method, e.g., rule curve [38–42], linear regression [43–48], artificial neural network [49–52], neuro-fuzzy inference system [53, 54], decision tree [50, 55], genetic programming [56], and support vector regression [57]. Compared with ISO, ESO directly operates on probabilistically described inflows [58–62]. The deterministic optimization is usually nonlinear, nonconvex, and nondifferentiable [63–66] and has been extensively addressed by metaheuristics recently [67].

Integrated GPUs are prevalent nowadays and can be found in both laptop (e.g., the IHDG520 GPU and the Intel HD Graphics 620 GPU) and desktop computers (e.g., the Intel HD Graphics 530 GPU). The clock rates of the Intel HD Graphics 520, 620, and 530 GPUs are all rather low, being 0.3 GHz, 0.3 GHz, and 0.35 GHz, respectively. The Intel HD Graphics 620 and 530 GPUs also lack dedicated graphics memory. The Intel integrated GPUs feature significantly different architectures from NVIDIA and AMD GPUs studied in the existing literature body, e.g., the NVIDIA GPUs studied in [1–4, 6–10, 12, 21, 22, 25, 26, 30] and the AMD GPUs studied in [13, 14, 68, 69] have much higher clock rates and have dedicated graphics memories. Many NVIDIA and AMD GPUs are quite larger in size and require higher power supply. The motherboard and the power supply of a brand desktop computer often differ considerably from those of a self-assembled desktop computer and may not support adding an NVIDIA or AMD GPU. As a result, this paper experimenting on the IHDG520 GPU is of critical practical meaning. In addition, to the best of our knowledge, this paper is the first pioneering work investigating parallelizing the deterministic optimization for the ISO of a reservoir system on a GPU.

The rest of this paper is organized as follows. The working procedure of CLPSO, the knowledge about OpenCL, and the characteristics of the IHDG520 GPU are detailed in Section 2. Section 3 introduces the case study of the Xiaowan Reservoir and formulates the deterministic optimization for the ISO of the Xiaowan Reservoir. Section 4 presents our proposed coarse-grained all-GPU and multipopulation models of CLSPO implemented by OpenCL. The performance of the models is evaluated in Section 5. Section 6 concludes this paper.

2. Background

2.1. Comprehensive Learning Particle Swarm Optimization. Let the search space be D-dimensional, and there are N particles in the swarm. Each particle, denoted as i ($i = 1, 2, \ldots, N$), is associated with velocity $V_i = V_{i,1}, V_{i,2}, \ldots, V_{i,D}$ and a position $P_i = P_{i,1}, P_{i,2}, \ldots, P_{i,D}$. In each iteration (or generation), V_i and P_i are updated on each dimension d ($d = 1, 2, \ldots, D$) as follows:

$$V_{i,d} = wV_{i,d} + 1.5\delta_{i,d}\left(E_{i,d} - P_{i,d}\right), \tag{1}$$

$$P_{i,d} = P_{i,d} + V_{i,d}, \tag{2}$$

where w is the inertia weight; $\delta_{i,d}$ is a random number uniformly distributed in [0, 1]; and $E_i = E_{i,1}, E_{i,2}, \ldots, E_{i,D}$ is the exemplar that guides the update of i's flight trajectory.

The dimensional velocity $V_{i,d}$ is clamped to a positive value $\overline{V_d}$, i.e.,

$$V_{i,d} = \begin{cases} \overline{V_d}, & \text{if } V_{i,d} > \overline{V_d}, \\ -\overline{V_d}, & \text{else if } V_{i,d} < -\overline{V_d}, \\ V_{i,d}, & \text{otherwise.} \end{cases} \tag{3}$$

Let $\underline{P_d}$ and $\overline{P_d}$, respectively, be the lower bound and the upper bound of the search space on each dimension d, $\overline{V_d} = 0.2(\overline{P_d} - \underline{P_d})$. At the beginning of the run of CLPSO, $V_{i,d}$ is initialized as a random value uniformly distributed in $[-\overline{V_d}, \overline{V_d}]$, and the dimensional position $P_{i,d}$ is initialized as a random value uniformly distributed in $[\underline{P_d}, \overline{P_d}]$.

The weight w linearly decreases from 0.9 to 0.4 in each generation k according to the following equation:

$$w = 0.9 - 0.5\frac{k}{K}, \tag{4}$$

where K is the predefined number of generations.

CLPSO maintains a personal best position $B_i = B_{i,1}, B_{i,2}, \ldots, B_{i,D}$ for each particle i. Initially, $B_i = P_i$. Let f be the fitness function. In each generation, if i's fitness value $f(P_i)$ is better than i's personal best fitness value $f(B_i)$, then $B_i = P_i$; otherwise, B_i does not change. The dimensional exemplar $E_{i,d}$ can be $B_{i,d}$ or $B_{j,d}$ with $j \neq i$, and the choice depends on i's learning probability L_i. To be specific, the value of L_i is

$$L_i = 0.05 + 0.45\frac{\exp(10(i-1/N-1))-1}{\exp(10)-1}. \tag{5}$$

On each dimension d, a random number uniformly distributed in [0, 1] is generated. If the generated number is no less than L_i, $E_{i,d} = B_{i,d}$; otherwise, $E_{i,d} = B_{j,d}$. To determine j, two different particles excluding i are randomly selected, and j is the particle with a better personal best fitness value. If $E_i = B_i$ on all the dimensions, CLPSO randomly chooses one dimension d and one particle j with $j \neq i$ to force $E_{i,d} = B_{j,d}$. The exemplar E_i is not updated unless $f(B_i)$ ceases improving for a refreshing gap of 7 generations. In each generation, $f(P_i)$ is evaluated only if P_i is feasible, i.e., $P_{i,d}$ is within $[\underline{P_d}, \overline{P_d}]$ on each dimension d. As E_i is always feasible, infeasible P_i will eventually be drawn back to the search space. After K generations, CLPSO determines the global best position $G = G_1, G_2, \ldots, G_D$ that exhibits the best fitness value among all the personal best positions. $f(G)$ is the swarm's global best fitness value. The step-by-step flowchart of CLPSO is depicted in Figure 1. If N is large enough, the chances of selecting two same particles and $j = i$ at Step 3 are very small; hence, the procedure to determine j can be simplified and just slightly affect the performance of CLPSO by just randomly selecting two particles when the generated random number is greater than L_i and randomly selecting a particle when $E_i = B_i$ on all the dimensions.

2.2. Open Computing Language. As can be seen from Figure 2, OpenCL views the hardware platform as a collection of heterogeneous compute devices attached to and managed by a CPU. A compute device can be a GPU or some other type of processor. OpenCL implements parallel computing with a host program and kernels. The host program runs on the CPU and can launch a kernel on a compute device. The parallel threads of a kernel are termed as work items. The work items can be organized into a number of independent so-called work groups, and only the work items executed on the same compute unit can be included in one work group. Each work item is identified by a unique global ID. The index space of the global IDs is one, two, or three-dimensional, with each attribute starting at zero. Each work item can also be identified by the group ID of the work group and the local ID of the work item relative to the work group.

The host program creates a context. A context specifies kernel (s) to be executed on one or more compute devices. Besides kernel, a context also manages objects such as command queue, memory, and program. A command queue holds commands (or operations) that will be executed on a compute device. Commands placed into a command queue can be classified into three categories, i.e., kernel management, memory management, and synchronization. Values for the input parameters of a kernel are transferred between the CPU and a compute device. OpenCL represents generic data by a buffer and supports creating a buffer only for a one-dimensional array. The memory space of a compute device is divided into four regions, i.e., global memory, constant memory, local memory, and private memory. The global memory permits read/write access to all the work items in all the work groups. Being writable by the CPU but not the compute device, the constant memory remains constant during the execution of a kernel. A local memory is just shared by all the work items in one specific work group. Each work item has a private memory, invisible by any other work item. Memory region qualifiers, "__global," "__constant," "__local," and "__private," can be applied on an input parameter of a kernel to, respectively, restrict that the parameter is to be stored in the global memory region, the constant memory region, the local memory region, or the private memory region. An input parameter with no memory region qualifier is stored in the private memory region by default. An input parameter with the data transferred by a buffer can only be stored in the global memory region. All the variables and constants additionally declared inside the kernel are stored in the private memory region. OpenCL is able to synchronize all the work items in the same work group, but cannot synchronize work items across different work groups. A program consists of one or more kernels.

2.3. Intel HD Graphics 520 Graphical Processing Unit. IHDG520 is an integrated GPU, i.e., it is embedded on the same die as the CPU. Integrated GPUs lead to less heat output and less power usage; thus, they have been widely taken in laptop (particularly, ultrabook) computers. The IHDG520 GPU has 24 compute units clocked at 0.3 GHz. Each compute unit is composed of 256 processing elements. The IHDG520 GPU has to share the main memory with the CPU. For the IHDG520 GPU, the size of the constant memory region and that of the local memory region are both zero; in other words, only the global memory region and the private memory region located in the main memory can be used. The size of the global memory region is 1.3 GB. The maximum size of a buffer created in the global memory region is 511 MB. The IHDG520 GPU uses on-chip registers to store kernel instructions. The IHDG520 GPU supports single-precision floating point calculation, but does not support double-precision floating point calculation.

3. Case Study and Deterministic Optimization Problems' Formulation

3.1. Case Study. The Xiaowan Reservoir located on Lancang River is taken as the case for study. Lancang River is the upper stream of Mekong River in China. Mekong River is a cross-border river in Southeast Asia. Originating from the

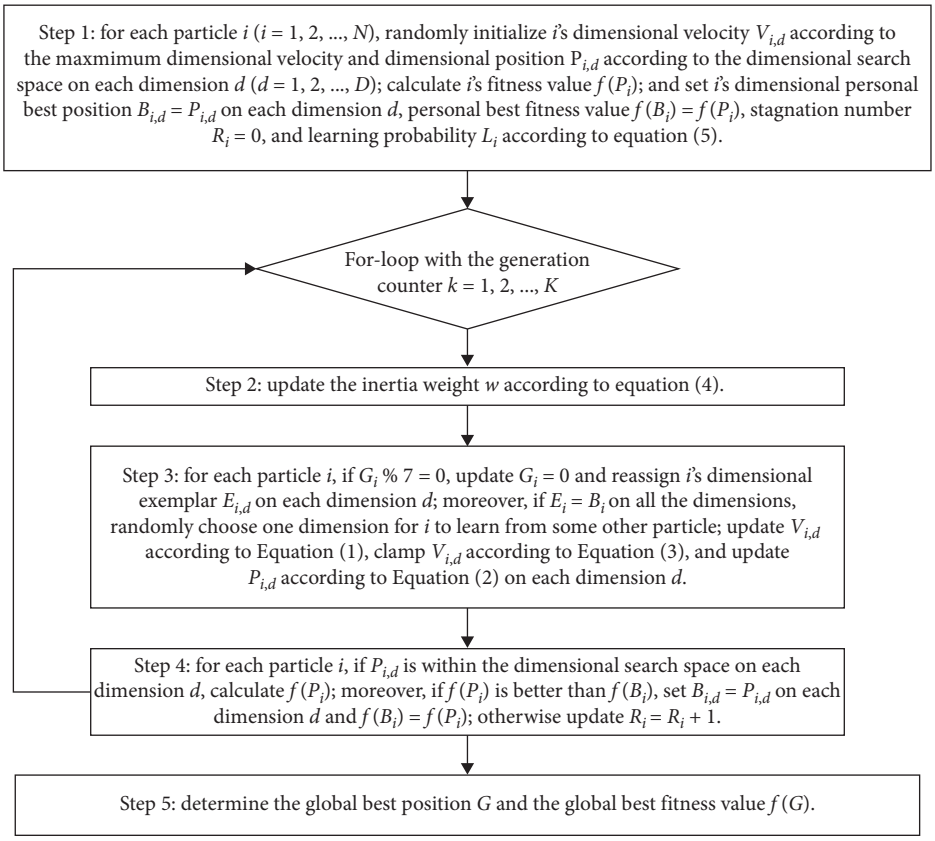

Figure 1: Flowchart of CLPSO.

Qinghai-Tibet Plateau, Mekong River runs through 6 countries, i.e., China, Myanmar, Laos, Thailand, Cambodia, and Vietnam, sequentially. Mekong River is the world's 12th longest river, with a length of 4350 km. The length of Lancang River is 2139 km, draining an area of 0.16 million km^2 over the provinces including Qinghai, Tibet, and Yunnan. The Xiaowan Reservoir is constructed at the west of Yunnan and on the middle reach of Lancang River, with a longitude of 100°05′28″ and a latitude of 24°42′18″. Figure 3 shows the Lancang River basin in Yunnan and the Xiaowan Reservoir. The Xiaowan Reservoir is mainly used for hydropower generation and also serves flood control, irrigation, sediment control, and navigation. For the Xiaowan Reservoir, the installed power generation capacity is 4200 MW, the normal forebay elevation is 1240 m, the dead forebay elevation is 1166 m, the flood control forebay elevation is 1236 m, and the total storage volume is 14,914 million m^3. The Xiaowan Reservoir is affected by a monsoon climate, and the inflows feature seasonal variations. The flood season is from June to September. The guaranteed hydropower generation per year is 190·10^8 kWh. Historical inflow records for the Xiaowan Reservoir from the year 1953 to 2014 are available.

3.2. Deterministic Optimization Problems' Formulation. The deterministic optimization problems for the ISO of the Xiaowan Reservoir are formulated with a yearly planning horizon of 12 monthly time steps and an ensemble of M years. Assuming that, for each year m (m = 1, 2, . . ., M), the inflow $I_{m,t}$ into the reservoir in each month t (t = 1, 2, . . ., 12) of year m is already known, the deterministic optimization problem related to year m tries to determine the power discharge rate $Q_{m,t}$ and the spillage rate $S_{m,t}$ in each month t of year m for the objective of maximizing the total hydropower generation over the yearly planning horizon of year m; $I_{m,t}$, $Q_{m,t}$, and $S_{m,t}$ are all measured by the unit of m^3/s, and the following equation gives the objective:

$$\max \sum_{t=1}^{12} 24 U_{m,t} X_{m,t}, \tag{6}$$

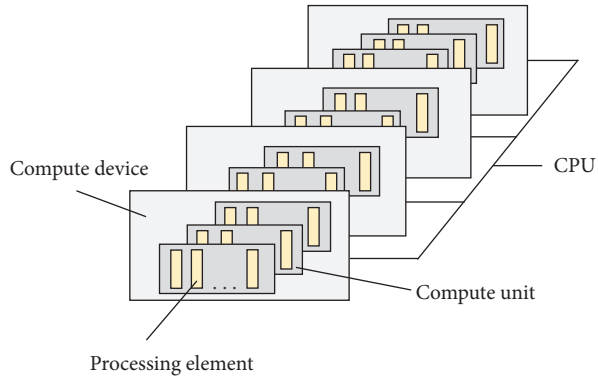

Figure 2: View of the hardware platform by OpenCL.

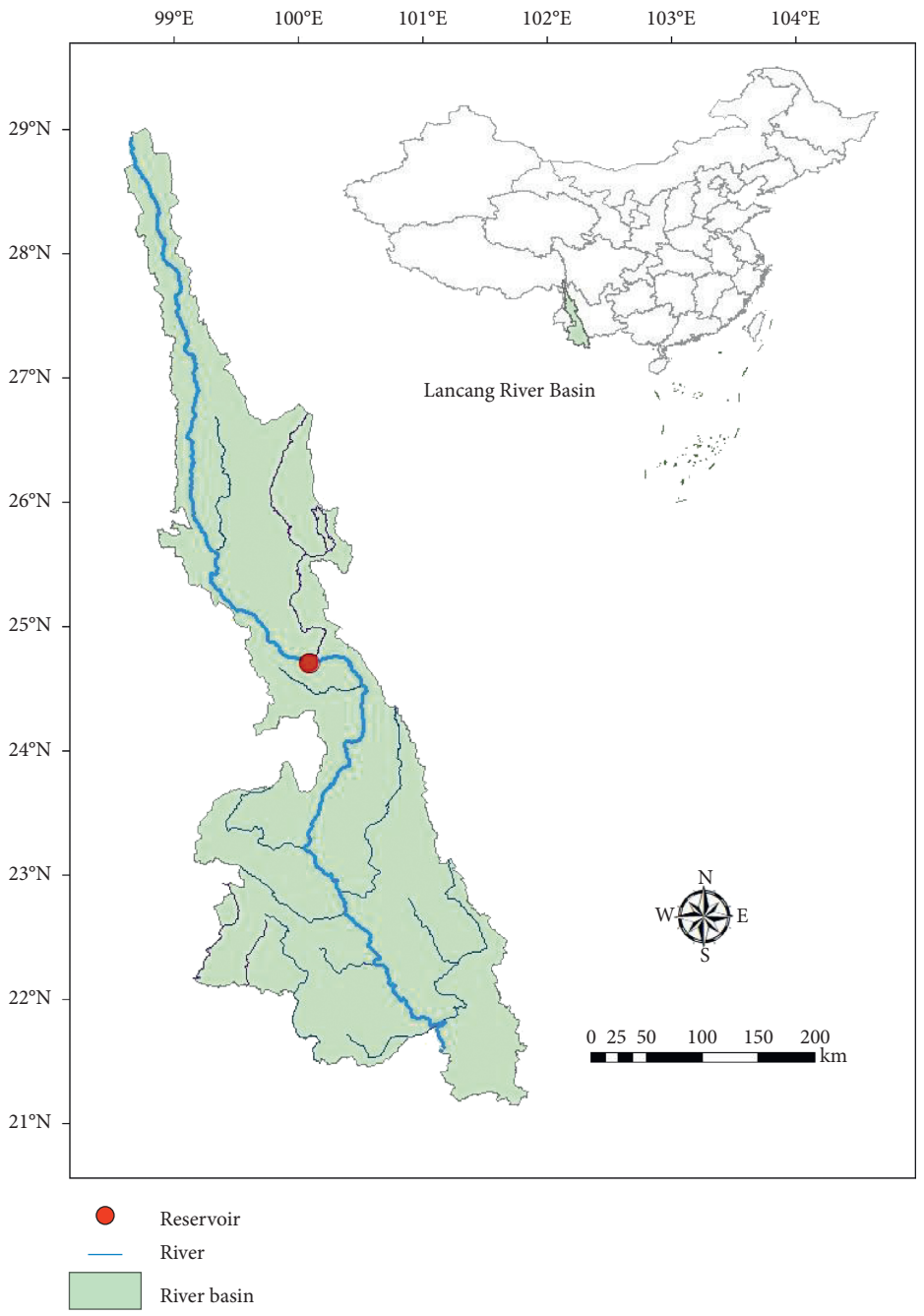

Figure 3: Lancang River basin in Yunnan and the Xiaowan Reservoir.

where $U_{m,\,t}$ is the power output in month t of year m and is measured by the unit of kW and $X_{m,\,t}$ is the number of days in month t of year m.

$U_{m,\,t}$ is calculated by

$$U_{m,t} = \frac{3600 Q_{m,t}}{C_{m,t}}, \tag{7}$$

where $C_{m,\,t}$ is the water conversion rate in month t of year m and is measured by the unit of m^3/kWh. $C_{m,\,t}$ is affected by the water head $H_{m,\,t}$ in month t of year m. $H_{m,\,t}$ is the difference of the forebay elevation $Y_{m,\,t}$ and the tailrace elevation $Z_{m,\,t}$ in month t of year m, i.e.,

$$H_{m,t} = Y_{m,t} - Z_{m,t}. \tag{8}$$

Let $A_{m,\,t}$ be the storage volume at the beginning of month t of year m. $Y_{m,\,t}$ is a function of the average storage volume $(A_{m,t} + A_{m,t+1})/2$ in month t of year m. $Z_{m,t}$ is a function of the outflow rate $O_{m,t}$ in month t of year m, and $O_{m,t} = Q_{m,t} + S_{m,t}$.

$Q_{m,t}$ cannot surpass the power discharge rate upper bound $\overline{Q_{m,t}}$ in month t of year m. As a result, $Q_{m,t}$ takes a smaller value of $O_{m,t}$ and $\overline{Q_{m,t}}$, as expressed in the following equation:

$$Q_{m,t} = \min\left\{O_{m,t}, \overline{Q_{m,t}}\right\}. \tag{9}$$

$\overline{Q_{m,t}}$ is a function of $H_{m,\,t}$.

Let $\widetilde{A}$ be the initial/final storage volume bound. The initial storage volume $A_{m,1}$ is known, and $A_{m,1} = \widetilde{A}$. The storage volume at the end of each month is calculated based on water balance, i.e.,

$$A_{m,t+1} = A_{m,t} + 86400\left(I_{m,t} - O_{m,t}\right)X_{m,t}, \quad t = 1, 2, \ldots, 12. \tag{10}$$

The problem is associated with the following constraints:

$$\underline{O_t} \le O_{m,t} \le \overline{O_t}, \quad t = 1, 2, \ldots, 12, \tag{11}$$

$$\underline{A_{t+1}} \le A_{m,t+1} \le \overline{A_{t+1}}, \quad t = 1, 2, \ldots, 11, \tag{12}$$

$$A_{m,13} = \widetilde{A}, \tag{13}$$

where $\underline{O_t}$ and $\overline{O_t}$ are, respectively, the lower and upper bounds of the outflow rate in each month t and $\underline{A_{t+1}}$ and $\overline{A_{t+1}}$ are, respectively, the lower and upper bounds of the storage volume at the end of each month t.

The deterministic optimization needs to solve M problems, with one problem for each year m separately. Monthly operation rules can be abstracted from the deterministic optimization results of all the M problems.

4. Parallelizing Comprehensive Learning Particle Swarm Optimization

4.1. Basic Coarse-Grained All-GPU Model. A basic parallelization scheme is presented here and works as the basis of our proposed enhancement strategies. The basic parallelization scheme follows the all-GPU model and implements a single kernel. CLPSO needs to generate random numbers uniformly distributed in [0, 1] at Steps 1 and 3. An OpenCL program is composed of both the host part and the kernel part. OpenCL provides no built-in primitive for generating any kind of random number in the kernel part. We write an auxiliary inline function that the kernel function can invoke for generating a random unsigned integer number based on the multiplicative linear congruential (MLC) principle [70], i.e.,

$$\phi = (1103515245\phi + 12345)\%2147483647, \tag{14}$$

where ϕ is a random unsigned integer number and works as the seed for generating the next random unsigned integer number. A random float number δ uniformly distributed in [0, 1] is obtained by

$$\delta = \frac{1.0\phi}{2147483647.0}. \tag{15}$$

When the inline function is called, the function code gets directly inserted at the point of each function call, thereby shortening the function call overhead.

The all-GPU model is coarse-grained, with each particle mapped to a separate work item in a one-dimensional index space. Each work item is identified by the global ID. N, D, $V_{i,d}$, $P_{i,d}$, $E_{i,\,d}$, $\underline{P_d}$, $\overline{P_d}$, $\overline{V_d}$, w, $B_{i,\,d}$, L_i, $f(P_i)$, $f(B_i)$, G_d, $f(G)$, and ϕ_i are input parameters of the kernel function, while k and K are additionally declared as variables/constants inside the kernel function. ϕ_i is the seed for each particle/work item i to generate a random unsigned integer number. Buffers are created for and the memory region qualifier "__global" is applied on $V_{i,d}$, $P_{i,d}$, $E_{i,\,d}$, $\underline{P_d}$, $\overline{P_d}$, $\overline{V_d}$, w, $B_{i,\,d}$, L_i, $f(P_i)$, $f(B_i)$, G_d, $f(G)$, and ϕ_i; hence, they are stored in the global memory region. N, D, k, and K are stored in the private memory region. The detailed working procedures of the host part and the kernel part are illustrated in Figure 4. The CPU first needs to initialize numerical values for some input parameters including N, D, $\underline{P_d}$, $\overline{P_d}$, $\overline{V_d}$, and ϕ_i. The numerical values are then transferred from the CPU to the IHDG520 GPU before the kernel function executes. The work items execute concurrently, and each work item is just responsible for performing the operations related to the corresponding particle at Steps 1, 3, and 4. Only one prespecified work item executes Steps 2 and 5. When the kernel function finishes execution, G_d and $f(G)$ are transferred back to the CPU.

4.2. Enhancement Strategies. Two enhancement strategies, namely, generating and transferring random numbers from the CPU as well as reducing the number of instructions in the kernel, are employed to accommodate the characteristics of the IHDG520 GPU and the OpenCL APIs for the purpose of significantly shortening the execution time of the basic coarse-grained all-GPU model.

OpenCL provides no built-in primitive for generating any type of random number. In the basic coarse-grained all-GPU model, an auxiliary inline function is written to assist the kernel function generating random numbers based on the MLC principle. The MLC principle generates a random unsigned integer number based on an unsigned integer input, and a random float number uniformly distributed in [0, 1] can be obtained by dividing the random unsigned integer number timed with 1.0 over 2147483647.0. Most GPUs including the IHDG520 GPU are not good at dealing with the integer multiplication and modulation operations as well as the float division operation involved in the MLC random number generation process and need many clock cycles to execute such costly operations. In addition, the IHDG520 GPU is slow at execution because its clock rate is just 0.3 GHz. Step 1 of CLPSO randomly initializes each particle i's dimensional velocity $V_{i,d}$ and position $P_{i,d}$ on each dimension d and requires $2ND$ random numbers. At Step 3, a random number is compared with each particle i's learning probability L_i on each dimension d, two particles are randomly selected for determining the dimensional exemplar $E_{i,\,d}$ if the number is greater than L_i, and a dimension is randomly selected to learn from a particle that is also randomly selected when E_i equals to the personal best position B_i on all the dimensions; thus, maximally, $3\,((K-1)/7+1)\,ND + 2\,((K-1)/7+1)\,N$ random numbers are needed. Step 4 updates each particle i's dimensional velocity

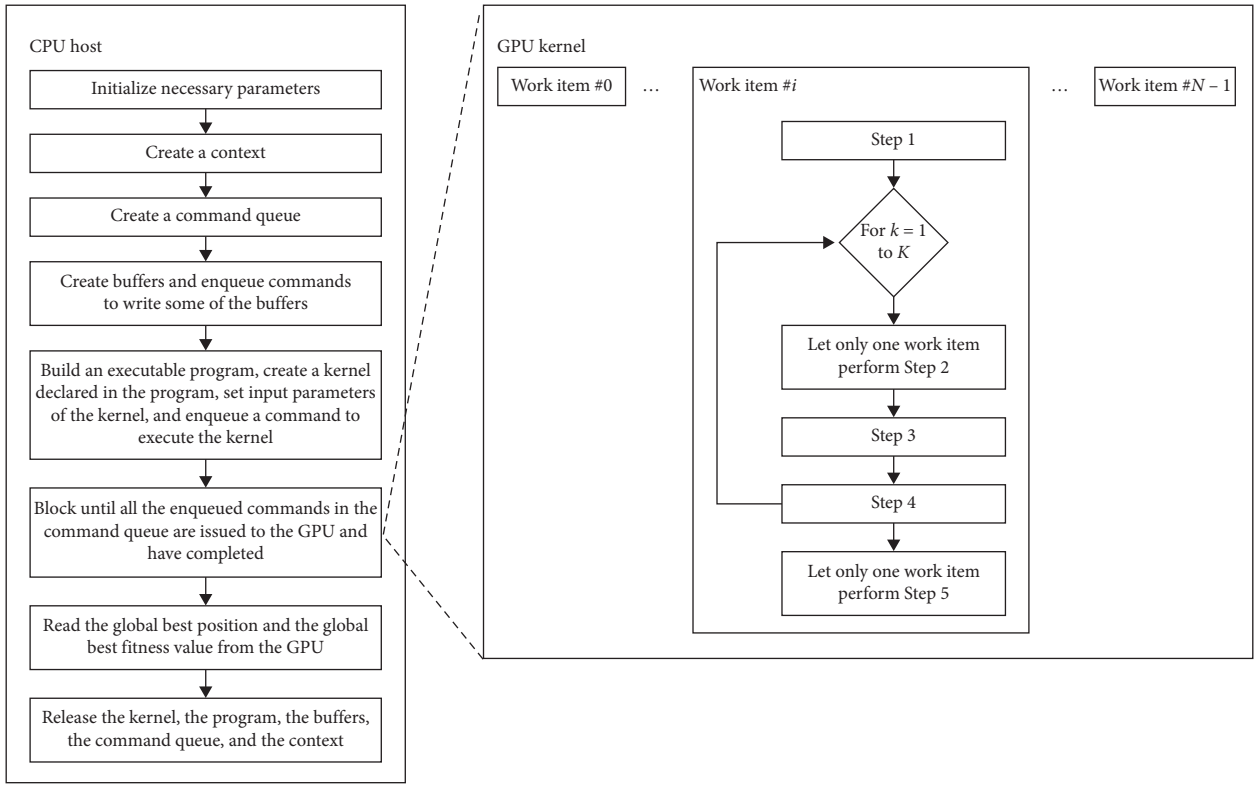

Figure 4: Basic coarse-grained all-GPU model.

$V_{i,d}$ with a random coefficient on each dimension d, and uses KND random numbers. It would be very time-consuming to generate all the $2ND + 3((K-1)/7+1)ND + 2((K-1)/7+1)N + KND$ random numbers on the IHDG520 GPU. The CPU is clocked at 2.5 GHz, and the host part of the parallelization program can invoke some highly efficient C/C++ library function to generate a random number uniformly distributed in [0, 1]. Therefore, we can generate all the random numbers on the CPU and transfer all the random numbers from the CPU to the IHDG520 GPU before the kernel function begins execution.

With respect to the basic coarse-grained all-GPU model, $V_{i,d}$, $P_{i,d}$, $E_{i,\,d}$, $B_{i,\,d}$, L_i, $f(P_i)$, and $f(B_i)$ are input parameters of the kernel function. OpenCL supports creating a buffer only for a one-dimensional array. Buffers representing one-dimensional arrays with ND elements are created for $V_{i,d}$, $P_{i,d}$, $E_{i,\,d}$, and $B_{i,\,d}$, and buffers representing one-dimensional arrays with D elements are created for L_i, $f(P_i)$, and $f(B_i)$. All the work items are indexed in a one-dimensional space, and the global IDs of the work items range from 0 to $N-1$, with each work item standing for a separate particle. The kernel function thus accesses $V_{i,d}$, $P_{i,d}$, $E_{i,\,d}$, and $B_{i,\,d}$ by the index $iN + d$ from the corresponding one-dimensional array and accesses L_i, $f(P_i)$, and $f(B_i)$ by the index i from the corresponding one-dimensional array, with i also being the global ID of the work item. $B_{i,\,d}$ and $f(B_i)$ are shared among the particles for exemplar redetermination at Step 3 of CLPSO, while $V_{i,d}$, $P_{i,d}$, $E_{i,\,d}$, L_i, and $f(P_i)$ are not shared by the particles. We can alternatively declare each particle's dimensional velocities, positions, and exemplars as one-dimensional arrays with D elements and also each particle's learning probability and fitness value as variables inside the kernel function, thereby reducing the number of addressing instructions. There are a number of if-clause conditional statements in the kernel function of the basic coarse-grained all-GPU model. Most GPUs including the IHDG520 GPU are also slow at performing conditional operations. Some if-clause conditional statements in the kernel function can actually be replaced by OpenCL built-in primitives. To be specific, clamping the dimensional velocity $V_{i,d}$ based on the maximum dimensional velocity $\overline{V_d}$ can be done by the primitive clamp $(V_{i,d}, -\overline{V_d}, \overline{V_d})$, and judging whether the dimensional position $P_{i,d}$ is feasible can be implemented as step $(P_{i,d}, \underline{P_d}) + \text{step}(\overline{P_d}, P_{i,d})$ through using the step primitive. The step primitive returns 0 if the right input parameter is less than the left input parameter, and 1, otherwise. The use of the OpenCL built-in primitives to replace some if-clause conditional statements therefore helps to shorten the time consumed on conditional operations.

4.3. Coarse-Grained Multipopulation Model. Like in [66], the storage volume constraints as expressed in equations (12) and (13) are repaired as follows.

First, for each year m, reversely starting from $t = 12$ and ending at 1, sequentially calculate the new lower bound $\underline{A_{m,t+1}}$ and upper bound $\overline{A_{m,t+1}}$ for the storage volume at the end of month t according to equations (16) and (17), respectively:

$$\underline{A_{m,t+1}} = \begin{cases} \tilde{A}, & \text{if } t = 12, \\ \max\left\{\underline{A_{m,t+2}} - 86400\left(I_{m,t} - O_{m,t}\right)X_{m,t}, \underline{A_{t+1}}\right\}, & \text{else,} \end{cases} \tag{16}$$

$$\underline{A_{m,t+1}} = \begin{cases} \tilde{A}, & \text{if } t = 12, \\ \min\left\{\overline{A_{m,t+2}} - 86400\left(I_{m,t} - O_{m,t}\right)X_{m,t}, \overline{A_{t+1}}\right\}, & \text{else.} \end{cases} \tag{17}$$

Second, incrementally starting from $t = 1$ and ending at 12, sequentially calculate $\Delta_{m,t}$ which is the deviation of the storage volume at the end of month t according to equation (18), modify $O_{m,t}$ according to equation (19), and update $A_{m,t+1}$ if $\Delta_{m,t} \neq 0$. Note that $O_{m,t}$ is kept feasible in equation (19).

$$\Delta_{m,t} = \begin{cases} A_{m,t+1} - \underline{A_{m,t+1}}, & \text{if } A_{m,t+1} < \underline{A_{m,t+1}}, \\ A_{m,t+1} - \overline{A_{m,t+1}}, & \text{else if } A_{m,t+1} > \overline{A_{m,t+1}}, \\ 0, & \text{otherwise,} \end{cases} \tag{18}$$

$$O_{m,t} = \begin{cases} O_t, & \text{if } \dfrac{O_{m,t} + \Delta_{m,t}}{\left(86400X_{m,t}\right)} < \underline{O_t}, \\ \overline{O_t}, & \text{else if } \dfrac{O_{m,t} + \Delta_{m,t}}{\left(86400X_{m,t}\right)} > \overline{O_t}, \\ O_{m,t} + \Delta_{m,t}, & \text{otherwise.} \end{cases} \tag{19}$$

Let $\Theta(A_{m,t+1})$ be the violation of the constraint for the storage volume at the end of month t. $\Theta(A_{m,t+1})$ is calculated according to the following equation:

$$\Theta\left(A_{m,t+1}\right) = \begin{cases} 1 - \dfrac{A_{m,t+1}}{\tilde{A}}, & \text{if } t = 12 \text{ and } A_{m,t+1} < \tilde{A}, \\ \dfrac{A_{m,t+1}}{\tilde{A}} - 1, & \text{else if } t = 12 \text{ and } A_{m,t+1} > \tilde{A}, \\ 1 - \dfrac{A_{m,t+1}}{\underline{A_{t+1}}}, & \text{else if } t < 12 \text{ and } A_{m,t+1} < \underline{A_{t+1}}, \\ \dfrac{A_{m,t+1}}{\overline{A_{t+1}}} - 1, & \text{else if } t < 12 \text{ and } A_{m,t+1} > \overline{A_{t+1}}. \end{cases} \tag{20}$$

The original constrained problem is converted to an unconstrained problem by optimizing the following objective that incorporates the violations:

$$\max \sum_{t=1}^{12} 24U_{m,t}X_{m,t} - \lambda \sum_{t=1}^{12} \Theta\left(A_{m,t+1}\right), \tag{21}$$

where λ is the penalty factor and is a large positive number. The term $\sum_{t=1}^{12} 24\,U_{m,t}X_{m,t}$ is the benefit, and the term $\lambda\sum_{t=1}^{12}\Theta(A_{m,t+1})$ is the violation cost.

The unconstrained problem is solved by CLPSO. Each particle's position is a 12-dimensional vector representing the outflow rates over the yearly planning horizon. The power discharge and spillage rates in each month can be easily determined from the outflow rate in the corresponding month. The deterministic optimization for ISO on the ensemble of M years is parallelized by a coarse-grained multipopulation model extended from the all-GPU model. The kernel part of the multipopulation model is illustrated in Figure 5. M work groups are used to concurrently tackle the M optimal operation problems, with each work group consisting of N work items and solving the optimal operation problem related to a different year following the all-GPU model. Each work group determines the global best position, global best fitness, global best benefit, and global best violation cost for the corresponding optimal operation problem. The global best benefit is the benefit of the global best position, and the global best violation cost is the violation cost of the global best position. The global best position, global best fitness, global best benefit, and global best violation cost results of all the work groups are transferred back from the GPU to the CPU when the kernel function finishes execution. By summing the global best benefit results and global best violation cost results, respectively, the CPU is able to obtain the total best benefit and the total best violation cost for the deterministic optimization. The summation can only be done by the host part on the CPU because OpenCL does not support synchronizing different work groups.

A serious challenge arises and needs to be addressed for the multipopulation model. The maximum size of a buffer created in the global memory region of the IHDG520 GPU is limited to be 511 MB, and the size of the global memory region is 1.3 GB. The multipopulation model needs maximally M $(2ND + 3$ $((K-1)/7+1)$ $ND + 2$ $((K-1)/7+1)$ $N + KND)$ random numbers uniformly distributed in [0, 1]. Suppose $N = 40$, $D = 12$, $K = 10000$, and $M = 62$, as a single-precision float number occupies 4 bytes, storing about 432 million random numbers requires a memory space of around 1.6 GB, exceeding the 511 MB limit for a buffer and the 1.3 GB size of the global memory region. We propose to create a buffer representing a one-dimensional array of $2ND + 3$ $((K-1)/7+1)$ ND+2 $((K-1)/7+1)$ $N + KND + M - 1$ random numbers. Each work item is identified by the group ID m and the local ID i simultaneously, with m ranging from 0 to $M-1$ and i ranging from 0 to $N-1$ in a one-dimensional index space. Each work item with the group ID m and the local ID i can access $2D + 3$ $((K-1)/7+1)D$+2 $((K-1)/7+1) + KD$ random numbers starting at the index $(2D$+3 $((K-1)/7+1)$ $D + 2$ $((K-1)/7+1) + KD)$ $i + m$ from the one-dimensional array stored in the buffer. This modified random number generation strategy aims to generate a small number of random numbers that can be flexibly used by the large number of work items without negatively impacting randomness.

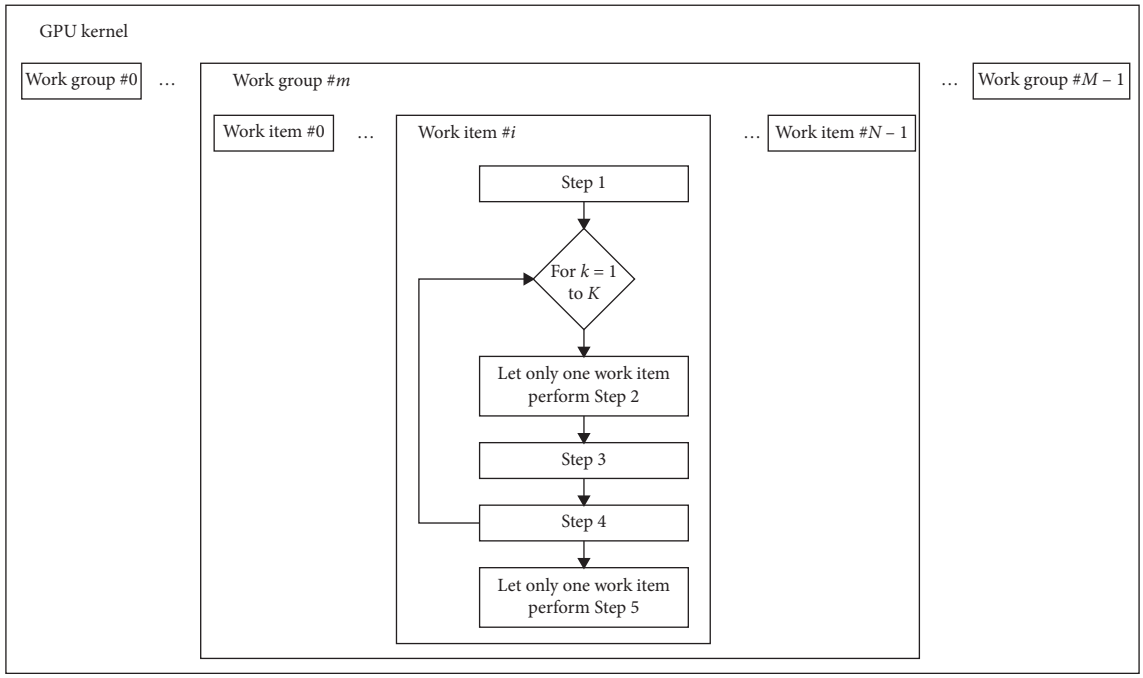

Figure 5: Coarse-grained multipopulation model.

5. Experimental Studies

5.1. Experimental Setup. In [27–29], the sequentialization implementation of CLPSO with a swarm of 40 particles was evaluated on various 30-dimensional benchmark global optimization functions for 25 runs. Four experimental issues are investigated in this paper: (1) how is the parallelization performance of the basic coarse-grained all-GPU model on all the benchmark functions? (2) how do the two enhancement strategies improve the parallelization performance of the basic all-GPU model on all the benchmark functions? (3) how is the parallelization performance of the multipopulation model extended from the all-GPU model on the case study of the deterministic optimization for the ISO of the Xiaowan Reservoir? and (4) would it be better to develop multiple kernels for parallelizing different phases of CLPSO?

For the 1st and 2nd issues, a sequentialization model of CLPSO and 3 coarse-grained all-GPU models as listed in Table 1 are implemented and compared. The sequentialization model involves a swarm of 40 particles. In all the 3 all-GPU models, 40 work items are used to concurrently execute the kernel on the IHDG520 GPU. Each particle/work item loops for 5000 generations. Eight 30-dimensional benchmark functions which were also used in [27–29] are evaluated. Table 2 gives the name, description, global optimum P^*, corresponding function value $f(P^*)$, and search space of each function. f_1, f_2, and f_3 are unimodal, and all the other functions are multimodal. f_7 and f_8 are rotated.

Regarding the 3rd issue, the multi population model is compared with the sequentialization model. The deterministic optimization for the ISO of the Xiaowan Reservoir is performed on the historical monthly inflow data recorded during the period of 62 years from 1953 to 2014. The optimal operation related to each year is solved by CLPSO with a swarm of 40 particles in the sequentialization model. The multipopulation model takes advantage of 62 work groups to concurrently tackle the 62 optimal operation problems. Each work group solves the optimal operation with respect to a separate year, is composed of 40 work items, and follows the final coarse-grained all-GPU model. Each work item iterates for 10,000 generations. The multipopulation model employs the modified random generation strategy. The initial/final storage volume bound is 145.57 10^8 m^3, corresponding to the normal forebay elevation as 1240 m. The penalty factor is $368*10^8$.

Concerning the 4th issue, we need to understand the overhead of kernel launching. The execution time of a parallelization model is the time gap between the initialization of parameters and the release of OpenCL objects and is the addition of CPU-side execution time and GPU-side execution time. The GPU-side execution time is the time spent on blocking until all the enqueued commands in the command queue are issued to the GPU and have completed. The CPU-side execution time can be divided into 6 segments: segment 1 is the time initializing the numerical values for some input parameters; segment 2 is the time creating a context, a command queue, and buffers, as well as enqueueing commands to write some of the buffers; segment 3 is the time building an executable program; segment 4 is the time creating a kernel declared in the program, setting input parameters of the kernel and enqueueing a command to execute the kernel; segment 5 is the time reading the results from the GPU and releasing the kernel; and segment 6 is the time releasing the other objects.

5.2. Experimental Results and Discussion. All the sequentialization and parallelization models are executed for 25 runs on all the benchmark functions and the case study. The

Table 1: Coarse-grained all-GPU models of CLPSO.

Model	Description
Basic	Parallelize CLPSO without employing any enhancement strategy
Intermediate	The basic model with the enhancement strategy generating and transferring random numbers from the CPU
Final	The intermediate model with the enhancement strategy reducing the number of instructions in the kernel

Table 2: Benchmark global optimization functions.

Function	Description	Global optimum P^*	$f(P^*)$	Search space
f_1	Sphere, $f_1(P) = \sum_{d=1}^{D} P_d^2$	$\{0\}^D$	0	$\{-100, 100\}^D$
f_2	Schwefel's P2.22, $f_2(P) = \sum_{d=1}^{D} \lvert P_d\rvert + \prod_{d=1}^{D} \lvert P_d\rvert$	$\{0\}^D$	0	$\{-10, 10\}^D$
f_3	Noise, $f_3(P) = \sum_{d=1}^{D} dP_d^4 + \text{random}[0, 1)$	$\{0\}^D$	0	$\{-1.28, 1.28\}^D$
f_4	Rosenbrock's, $f_4(P) = \sum_{d=1}^{D-1} (100(P_{d+1} - P_d^2) + (P_d - 1)^2))^2$	$\{1\}^D$	0	$\{-10, 10\}^D$
f_5	Rastrigin's, $f_5(P) = \sum_{d=1}^{D} (P_d^2 - 10\cos(2\pi P_d) + 10)$	$\{0\}^D$	0	$\{-5.12, 5.12\}^D$
f_6	Ackley's, $f_6(P) = -20\exp(-0.2\sqrt{(\sum_{d=1}^{D} P_d^2)/D}) - \exp((\sum_{d=1}^{D} (\cos 2\pi P_d))/D) + 20 + e$	$\{0\}^D$	0	$\{-32, 32\}^D$
f_7	Rotated Schwefel's, $f_7(\sigma) = 418.9828\,D - \sum_{d=1}^{D} \zeta_d$, where $\zeta_d = \begin{cases} \sigma_d \sin(\sqrt{\lvert\sigma_d\rvert}), & \lvert\sigma_d\rvert \le 500, \\ 0, & \text{otherwise}, \end{cases}$, $\sigma_d = \psi_d + 420.96$	$\{420.96\}^D$	0	$\{-500, 500\}^D$
f_8	Rotated Rastrigin's, $f_8(\sigma) = f_5(\sigma)$, where $\sigma = \Omega P$	$\{0\}^D$	0	$\{-5.12, 5.12\}^D$

Table 3: Statistical execution time and global best fitness results of the sequentialization model and the coarse-grained all-GPU models on all the benchmark functions.

Benchmark function	Model	Execution time (ms)				Global best fitness value			
		Mean	Standard deviation	Maximum	Minimum	Mean	Standard deviation	Maximum	Minimum
f_1	Sequentialization	167.24	7.17	172.00	156.00	0	0	0	0
	Basic	5823.84	16.12	5866.00	5803.00	0	0	0	0
	Intermediate	574.08	10.93	608.00	561.00	0	0	0	0
	Final	450.58	11.29	483.00	436.00	0	0	0	0
f_2	Sequentialization	192.84	7.70	203.00	187.00	0	0	0	0
	Basic	5881.24	12.77	5912.00	5865.00	0	0	0	0
	Intermediate	638.32	13.38	671.00	624.00	0	0	0	0
	Final	506.72	11.94	531.00	483.00	0	0	0	0
f_3	Sequentialization	164.72	7.90	172.00	156.00	$5.41E-3$	$1.34E-3$	$6.70E-3$	$3.18E-3$
	Basic	6148.96	28.44	6194.00	6084.00	$6.47E-3$	$2.04E-3$	$9.29E-3$	$2.04E-3$
	Intermediate	571.64	11.71	593.00	561.00	$5.95E-3$	$1.36E-3$	$8.76E-3$	$3.37E-3$
	Final	451.76	7.05	468.00	437.00	$5.12E-3$	$1.71E-3$	$9.14E-3$	$1.93E-3$
f_4	Sequentialization	161.64	7.68	172.00	156.00	32.54	14.89	54.60	21.19
	Basic	6007.28	14.83	6037.00	5975.00	36.61	18.11	82.75	10.87
	Intermediate	596.96	10.09	609.00	577.00	35.76	19.33	74.10	9.75
	Final	454.28	8.18	468.00	437.00	33.62	20.24	84.01	10.90
f_5	Sequentialization	254.60	7.47	266.00	249.00	$3.00E-6$	$3.00E-6$	$8.00E-6$	0
	Basic	5982.96	15.74	6022.00	5959.00	$1.08E-4$	$1.12E-4$	$4.60E-4$	$1.30E-5$
	Intermediate	621.52	7.42	640.00	608.00	$6.60E-5$	$4.80E-5$	$1.87E-4$	$6.00E-6$
	Final	487.36	10.36	515.00	468.00	$8.00E-6$	$1.10E-5$	$5.30E-5$	$1.00E-6$
f_6	Sequentialization	310.12	5.20	312.00	296.00	$1.10E-5$	$2.00E-6$	$1.50E-5$	$8.00E-6$
	Basic	5997.92	15.76	6038.00	5974.00	$1.60E-5$	$2.00E-6$	$1.90E-5$	$1.10E-5$
	Intermediate	646.68	11.11	671.00	624.00	$1.50E-5$	$1.00E-6$	$1.50E-5$	$1.10E-5$
	Final	514.16	10.62	531.00	499.00	$1.20E-5$	$2.00E-6$	$1.50E-5$	$8.00E-6$
f_7	Sequentialization	492.32	7.87	500.00	483.00	1289.46	181.45	1496.70	874.44
	Basic	3059.44	31.37	3105.00	2995.00	1279.78	153.06	1586.59	962.10
	Intermediate	746.92	10.50	765.00	733.00	1307.11	122.96	1601.35	1119.69
	Final	714.48	10.99	733.00	702.00	1309.49	158.23	1560.73	1042.22
f_8	Sequentialization	531.04	7.16	546.00	514.00	26.56	4.24	36.60	18.25
	Basic	6571.36	15.69	6599.00	6552.00	27.93	3.19	34.43	22.45
	Intermediate	934.08	14.53	967.00	920.00	28.76	4.60	38.60	20.65
	Final	594.68	12.17	624.00	577.00	28.43	3.50	33.49	21.56

speedup of a parallelization model as compared with the sequentialization model is the value calculated as the mean execution time of the parallelization model divided by that of the sequentialization model. Table 3 gives the statistical (i.e., mean, standard deviation, maximum, and minimum) execution time and global best fitness value results of the sequentialization model and the intermediate, basic, and final all-GPU models on all the functions. To determine whether the solutions obtained by the sequentialization model are statistically different from those obtained by the final all-GPU model, two-tailed t-tests with the assumption of equal variances and the significance level 0.05 are carried out, and the t-test results are listed in Table 4. Table 5 lists the statistical execution time, total best benefit, and total best violation cost results of the sequentialization model and the multipopulation model on the case study. The two-tailed t-test and speedup results from the comparison of the sequentialization model and the multipopulation model on the case study are listed in Table 6. For the Xiaowan Reservoir, year 1954 is a typical wet year with natural inflow 1528 m^3/s in average, 1976 is a typical normal year with natural inflow 1186 m^3/s in average, and 2011 is a typical dry year with natural inflow 974 m^3/s in average. The monthly natural inflow records as well as outflow rate, forebay elevation, and power output results determined from the median run by the multipopulation model for 1954, 1976, and 2011 are, respectively, shown in Figures 6–9. Table 7 gives the mean CPU-side execution time results of the final all-GPU model on some benchmark functions and the multipopulation model on the case study.

The original sequentialization implementation of CLPSO proposed in [27–29] also uses 40 particles that iterate for 5000 generations and was also evaluated on the 30-dimensional Sphere, Schewefel's P2.22, Noise, Rosenbrock's, Rastrigin's, Ackley's, rotated Schwefel's, and rotated Rastrigin's functions for 25 runs. Our sequentialization model described in Table 3 differs from the original sequentialization implementation in that the inequality conditions enforced on determining the dimensional exemplar are relaxed to facilitate generating and transferring random numbers from the CPU. The statistical global best fitness results of our sequentialization model as given in Table 3 are similar with those of the original sequentialization implementation found in [27–29], verifying that the relaxation of the inequality conditions causes little negative impact on the final solution's quality when the number of particles is large enough. The sequentialization model and the basic, intermediate, and final all-GPU models all use single-precision float numbers and directly report 0 for sufficiently small float numbers, e.g., the statistical global best fitness results of all the models are all 0 on functions f_1 and f_2 in Table 3.

As can be seen from Table 3, the statistical global best fitness results of the basic, intermediate, and final all-GPU models are the same as those of the sequentialization model on f_1 and f_2 and similar with those of the sequentialization model on the other functions. The t-test results reported in Table 4 indicate that the global best fitness results of the final model are statistically indifferent on f_3 to f_8 as the t-test results are greater than the significance level 0.05. A t-test cannot proceed when the two pairs of data samples are all 0; hence, the t-test results on f_1 and f_2 are blank. Functions f_1 to f_3 are unimodal, and f_4 to f_8 are multimodal. The statistical global best fitness results given in Table 3 show that the sequentialization model and all the all-GPU models of CLPSO can find the global optimum on f_1 and f_2 in all the runs, can find the global optimum on f_5 in some runs and a near-optimum in the other runs, can find a near-optimum on f_3 and f_6 in all the runs, and can only find a local optimum on f_5, f_7, and f_8 in all the runs.

Table 4: Two-tailed t-test results from the comparison of the sequentialization model and the final all-GPU model on all the benchmark functions.

Benchmark function	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8
Two-tailed t-test	—	—	0.40	0.11	0.08	0.20	0.37	0.89

For the sequentialization model, a significant part of the execution time is spent on fitness calculation. Functions f_5 and f_6 include cosine operations. Compared with f_5, f_6 additionally needs to calculate exponential values. f_7 and f_8 are rotated functions and multiply the original decision vector by an orthogonal matrix. f_8 is a rotated variant of f_5. There are sine operations in f_7. Therefore, the evaluation of f_7 and f_8 is most time-consuming followed by f_5 and f_6, and f_1 to f_4 are least time-consuming. The analysis is clearly validated by the statistical execution time results of the sequentialization model given in Table 3.

In Table 3, the basic, intermediate, and final all-GPU models exhibit similar statistical global best fitness results on all the functions. Owing to the ways adopting the enhancement strategies, the statistical execution time results of the 3 all-GPU models are quite different, particularly the basic model and the intermediate model. The mean execution time results of the basic model are around 6000 ms on $f_1, f_2, f_3, f_4, f_5, f_6$, and f_8 and around 3000 ms on f_7. The mean execution time results of the intermediate all-GPU model are around 600 ms on f_1 to f_6, around 700 ms on f_7, and around 900 ms on f_8, far less than those of the basic model. The dramatic execution time difference between the basic model and the intermediate model on the same function is attributed to random number generation. The basic model generates random numbers in the kernel function based on the MLC principle. Performing the integer multiplication and modulo as well as float division operations involved in the MLC random number generation process on the IHDG520 GPU is very time-consuming. In contrast, the intermediate model generates random numbers on the high clock rate CPU efficiently and transfers the random numbers from the CPU to the GPU. The basic model takes less time on f_7 than on the other functions because the landscape of f_7 is highly mountainous; each particle is likely to fly to a position that leads to a better personal best fitness value during the trajectory update, and the model thus goes through much less times of exemplar redetermination and invokes much less times of random number generation. The mean execution time results of the final model are around 100 ms less than those of the intermediate model on f_1 to f_6,

Table 5: Statistical execution time, total best benefit, and total best violation cost results of the sequentialization model and the coarse-grained multipopulation model on the case study.

Model	Execution time (ms)				Total best benefit (10^8 kWh)				Total best violation cost			
	Mean	Standard deviation	Maximum	Minimum	Mean	Standard deviation	Maximum	Minimum	Mean	Standard deviation	Maximum	Minimum
Sequentialization	16,090.00	25.39	16,177.00	16,053.00	14,550.50	0.42	14,550.80	14,548.85	0	0	0	0
Multipopulation	1165.60	9.52	1185.00	1139.00	14,550.42	0.23	14,550.72	14,550.04	0	0	0	0

Table 6: Two-tailed t-test and speedup results from the comparison of the sequentialization model and the multipopulation model on the case study.

Two-tailed t-test	0.38
Speedup	13.80

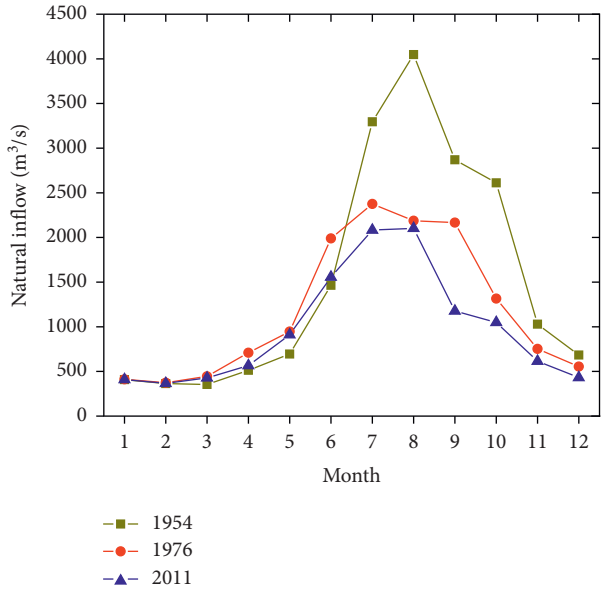

Figure 6: Natural inflow records of the three typical years.

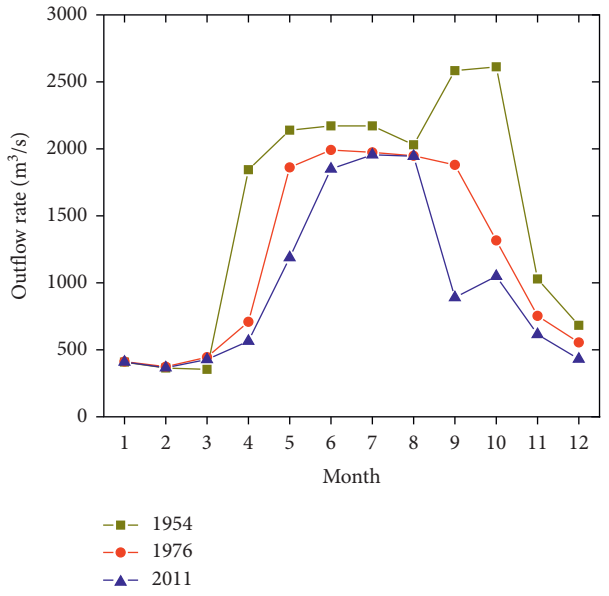

Figure 7: Outflow rate results of the three typical years.

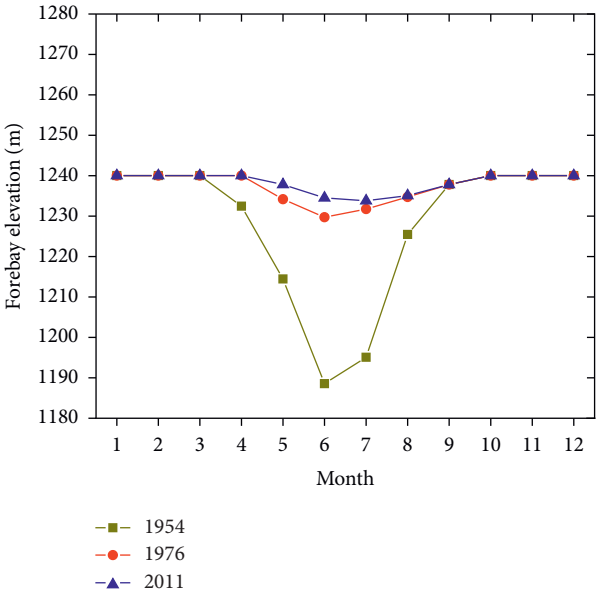

Figure 8: Forebay elevation results of the three typical years.

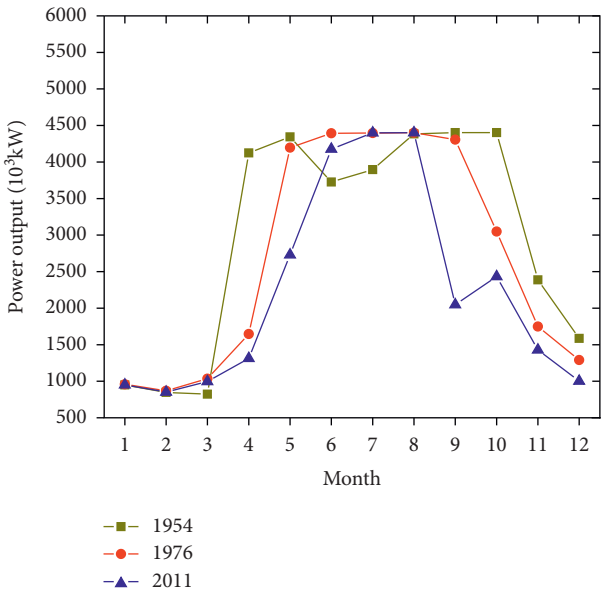

Figure 9: Power output results of the three typical years.

around 30 ms less on f_7, and around 300 ms less on f_8, verifying that the strategy reducing the number of instructions in the kernel benefits shortening the execution time. With respect to f_8, multiplication of the dimensional

Table 7: Mean CPU-side execution time results of the final all-GPU model on some benchmark functions and the multipopulation model on the case study.

Model	Benchmark function/case study	Mean CPU-side execution time (ms)						
		Segment 1	Segment 2	Segment 3	Segment 4	Segment 5	Segment 6	Total
Final all-GPU	f_1	134.16	18.96	171.24	0.60	0.60	2.96	328.52
	f_3	132.84	18.16	167.16	0.64	0.60	2.56	321.96
	f_7	133.40	18.55	277.16	0.60	0.58	2.79	433.08
Multipopulation	Case study	108.52	21.24	301.40	0.64	0.55	2.62	434.97

position $P_{i,d}$ by the orthogonal matrix is actually a two-layer nested for-loop; as a result, shortening of the execution time is much more on f_8 than on the other functions. Particles are likely to be infeasible when the work items concurrently optimize the highly mountainous function f_7; hence, shortening of the execution time is not that noticeable on f_7. The mean execution time results of the final model however are still more than those of the sequentialization model on all the functions; this is because a considerable amount of time (in the scale of hundreds of ms) must be spent on creating/releasing OpenCL objects (e.g., context, command queue, buffer, program, and kernel) and transferring buffers between the CPU and the GPU, as verified by the mean CPU-side execution time results given in Table 7. It can be seen from Table 3 that the standard deviation execution time results of the sequentialization model and all the all-GPU models are relatively small as compared with the corresponding mean execution time results, meaning that the execution time results of each model do not vary much in each run.

The deterministic optimization for the ISO of the Xiaowan Reservoir is multimodal, as reflected from the statistical total best benefit results of the sequentialization model listed in Table 5. The standard deviation total best benefit of the sequentialization model is 0.42 10^8 kWh. The statistical total best benefit results of the multipopulation model are similar with those of the sequentialization model. The t-test results given in Table 6 indicate that the total best benefit results of the two models are statistically indifferent. Accordingly, the modified random generation strategy does not hurt randomness. The mean total best benefit of the sequentialization model and that of the multipopulation model are around 14,550 10^8 kWh; hence, in average, the optimized hydropower generation is about 235 10^8 kWh per year, much more than the guaranteed hydropower generation 190 10^8 kWh per year, validating the powerful global optimization capability of CLPSO. The solutions are feasible as the statistical total best violation cost results of the 2 models are all 0. The sequentialization model is very time-consuming, and its mean execution time is 16,090.00 ms. As we can see from Tables 5 and 6, the mean execution time of the multipopulation model is 1165.60 ms which is significantly less than that of the sequentialization model, and the speedup is 13.80. The significant speedup is achieved by parallelizing the 62 optimal operation problems and parallelizing the 40 particles for each optimal operation problem. The 2 models are robust in terms of execution time results in all the runs as the standard deviation execution time results are small.

It can be observed from Figure 6 that the natural inflows are large in the flood season from June to September for all the 3 typical years. The natural inflows of year 1954 are considerably more than those of 1976 in July, August, September, and October, and those of 1976 are noticeably more than those of 2011 in June, July, September, and October. As Figures 7–9 show, the Xiaowan Reservoir needs to release much more outflows in April, May, June, July, September, October, and November of 1954 than in the corresponding months of 1976, leading to much lower forebay elevations in April, May, June, July and August, less power outputs in March, June, and July, and more power outputs in April, October, November, and December. The outflow rates and power outputs of 1976 are more than those of 2011 in April, May, June, September, October, November, and December.

As we can see from Table 7, the mean segment 1 time, segment 2 time, segment 4 time, segment 5 time, and segment 6 time results of the final all-GPU model are similar on functions f_1, f_3, and f_7, with the mean segment 1 time results more than, the mean segment 2 time results less than, and the mean segment 4, 5, and 6 time results similar to the mean segment 1 time result, the mean segment 2 time result, and the mean segment 4, 5, and 6 time results of the multipopulation model on the case study, respectively. The segment 1 time is mainly the time generating the random numbers and is thus affected by the number of random numbers. The segment 2 time increases with the number of buffers created. The mean segment 3 time results of the final all-GPU model on f_1 and f_3 are similar and are much less than those of the final all-GPU model on f_7 and the multipopulation model on the case study, indicating that the more difficult fitness evaluation of a particle, the more time building the program. Steps 1, 3, and 4 of CLPSO involve operations related to each work item, while Steps 2 and 5 are executed by just one prespecified work item. Steps 2, 3, and 4 constitute a for-loop. When multiple kernels are used for parallelizing different phases of CLPSO, intermediate results must be transferred back from a kernel, and if the kernel is not the last kernel, then the intermediate results need to be transferred to the next kernel. Steps 2, 3, and 4 cannot be implemented as multiple kernels as the for-loop causes the overhead of frequently enqueueing commands to write some buffers, enqueueing commands to execute the kernels, and reading results from the kernels. The overhead can be much large with respect to many generations. Steps 1 and 5 also do not benefit from being split as multiple kernels because all the work items are occupied at Step 1, and for the

multipopulation model, each work group has one work item occupied at Step 5. An alternative is to implement 3 kernels, respectively, corresponding to Step 1, the for-loop, and Step 5. The alternative incurs a small overhead of enqueueing commands to write some buffers, creating kernels, setting input parameters of the kernels, enqueueing commands to execute the kernels, reading results from the kernels, and releasing the kernels. Accordingly, our proposed final all-GPU model and multipopulation model for parallelizing CLPSO are appropriate.

6. Conclusions

In this paper, we have studied parallelizing CLPSO by OpenCL on the integrated IHDG520 GPU. We have firstly proposed a basic coarse-grained all-GPU model, with one kernel written and each work item representing a separate particle. As the IHDG520 GPU features a low clock rate and the CPU has a high clock rate, two strategies, i.e., generating and transferring random numbers from the CPU to the GPU as well as reducing the number of instructions in the kernel, have been adopted to shorten the basic model's execution time. To facilitate parallelization implementation of CLPSO, the inequality conditions used when determining a dimensional exemplar are relaxed. We have also studied a real-world case parallelizing the deterministic optimization for the ISO of the Xiaowan Reservoir. The deterministic optimization has been solved by CLPSO on 62 years' monthly natural inflow records and has been parallelized by a multipopulation model using a large number of work items extended from the all-GPU model. Owing to the size limits for a buffer transferring data from the CPU to the GPU and for storing the data in the global memory region, the random number generation strategy has been further modified by generating a small number of random numbers that can be flexibly exploited by the large number of work items without harming randomness. Experiments have been conducted on various unimodal/multimodal 30-dimensional benchmark global optimization functions and the case study. The experimental results demonstrate that (1) the relaxation of the inequality conditions causes little negative impact on the final solution's quality; (2) the two enhancement strategies help improve the basic model's efficiency; (3) the modified random number generation strategy is suitable for the case of a large number of work items; and (4) the multi population model is able to achieve the consumption of significantly less execution time than the corresponding sequentialization model. In the future, we will investigate adapting and applying the proposed models for parallelizing more advanced metaheuristics [29, 71–74] and solving more real-world large-scale problems.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was financially supported by the National Natural Science Foundation of China Projects (61703199, 61866023, and 61865012), the Shaanxi Province Natural Science Foundation Basic Research Project (2020JM-278), and the Central Universities Fundamental Research Foundation Project (GK202003006).

References

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

[2] V. Roberge and M. Tarbouchi, "Efficient parallel particle swarm optimizers on GPU for real-time harmonic minimization in multilevel inverters," in *Proceedings of the Annual Conference on IEEE Industrial Electronics Society,* IEEE, Montreal, Canada, October 2012.

[3] J. Platos, V. Snasel, T. Jezowicz et al., "A PSO-based document classification algorithm accelerated by the CUDA platform," in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics,* IEEE, Seoul, South Korea, October 2012.

[4] J. Zhao, W. Wang, W. Pedrycz, and X. Tian, "Online parameter optimization-based prediction for converter gas system by parallel strategies," *IEEE Transactions on Control Systems Technology*, vol. 20, no. 3, pp. 835–845, 2012.

[5] 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 *Proceedings of the Annual Conference Companion on Genetic and Evolutionary Computation*, July 2012.

[6] J. Reguera-Salgado and J. Martín-Herrero, "High performance GCP-based particle swarm optimization of orthorectification of airborne pushbroom imagery," in *Proceedings of the IEEE International Geoscience and Remote Sensing Symposium,* IEEE, Munich, Germany, July 2012.

[7] S. E. Papadakis and A. G. Bakrtzis, "A GPU accelerated PSO with application to Economic Dispatch problem," in *Proceedings of theInternational Conference on Intelligent System Application to Power Systems,* IEEE, Hersonissos, Greece, September 2011.

[8] O. Kilic, E. El-Araby, Q. Nguyen, and V. Dang, "Bio-inspired optimization for electromagnetic structure design using full-wave techniques on GPUs," *International Journal of Numerical Modelling: Electronic Networks, Devices and Fields*, vol. 26, no. 6, pp. 649–669, 2013.

[9] A. Ouyang, Z. Tang, X. Zhou, Y. Xu, G. Pan, and K. Li, "Parallel hybrid PSO with CUDA for lD heat conduction equation," *Computers & Fluids*, vol. 110, pp. 198–210, 2015.

[10] D. L. Souza, O. N. Teixeira, D. C. Monteiro et al., "A new cooperative evolutionary multi-swarm optimizer algorithm based on CUDA architecture applied to engineering optimization," in *Combinations of Intelligent Methods and Applications*, pp. 95–115, Springer, Berlin, Germany, 2013.

[11] B. Sharma, R. K. Thulasiram, and P. Thulasiraman, "Portfolio management using particle swarm optimization on GPU," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing with Applications,* IEEE, Leganes, Spain, July 2012.

[12] B. Sharma, R. K. Thulasiram, and P. Thulasiraman, "Normalized particle swarm optimization for complex chooser

option pricing on graphics processing unit," *The Journal of Supercomputing*, vol. 66, no. 1, pp. 170–192, 2013.

[13] D. Tristram, D. Hughes, and K. Bradshaw, "Accelerating a hydrological uncertainty ensemble model using graphics processing units (GPUs)," *Computers & Geosciences*, vol. 62, pp. 178–186, 2014.

[14] M. Abdelaziz, "GPU-opencl accelerated probabilistic power flow analysis using monte-carlo simulation," *Electric Power Systems Research*, vol. 147, pp. 70–72, 2017.

[15] I. Boussaïd, J. Lepagnot, and P. Siarry, "A survey on optimization metaheuristics," *Information Sciences*, vol. 237, pp. 82–117, 2013.

[16] Y. Tan and K. Ding, "A survey on GPU-based implementation of swarm intelligence algorithms," *IEEE Transactions on Cybernetics*, vol. 46, no. 9, pp. 2028–2041, 2016.

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

[18] Y.-D. Zhang, S.-H. Wang, and G.-L. Ji, "A comprehensive survey on particle swarm optimization algorithm and its applications," *Mathematical Problems in Engineering*, vol. 2015, Article ID 931256, 38 pages, 2015.

[19] Khronos Group, "OpenCL overview," 2019, https://www.khronos.org/opencl/.

[20] Nvidia Corporation, "CUDA toolkit," 2019, https://developer.nvidia.com/cuda-toolkit.

[21] Y. Hung and W. Wang, "Accelerating parallel particle swarm optimization via GPU," *Optimization Methods and Software*, vol. 27, no. 1, pp. 33–51, 2012.

[22] M. P. Wachowiak and A. E. L. Foster, "GPU-based asynchronous global optimization with particle swarm," *Journal of Physics: Conference Series*, vol. 385, 2012.

[23] J. Kennedy and R. Mendes, "Population structure and particle swarm performance," in *Proceedings of the IEEE Congress on Evolutionary Computation*, IEEE, Brisbane, Australia, June 2002.

[24] Particle Swarm Central, "Standard PSO 2006," 2006, http://www.particleswarm.info/Standard_PSO_2006.c.

[25] Y. Zhou and Y. Tan, "GPU-based parallel particle swarm optimization," in *Proceedings of the IEEE Congress on Evolutionary Computation*, IEEE, Trondheim, Norway, May 2009.

[26] S. Cagnoni, A. Bacchini, and L. Mussi, "OpenCL implementation of particle swarm optimization: a comparison between multi-core CPU and GPU performances," in *Proceedings of the European Conference on the Applications of Evolutionary Computation*, Springer, Málaga, Spain, April 2012.

[27] J. J. Liang, A. K. Qin, P. N. Suganthan, and S. Baskar, "Comprehensive learning particle swarm optimizer for global optimization of multimodal functions," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 3, pp. 281–295, 2006.

[28] Z.-H. Zhan, J. Zhang, Y. Li, and Y.-H. Shi, "Orthogonal learning particle swarm optimization," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 6, pp. 832–847, 2011.

[29] X. Yu and X. Zhang, "Enhanced comprehensive learning particle swarm optimization," *Applied Mathematics and Computation*, vol. 242, pp. 265–276, 2014.

[30] M. Rabinovich, "Particle swarm optimization on a GPU," in *Proceedings of the IEEE International Conference on Electro/Information Technology*, IEEE, Brookings, SD, USA, May 2012.

[31] S. Simonovic, "The implicit stochastic model for reservoir yield optimization," *Water Resources Research*, vol. 23, no. 12, pp. 2159–2165, 1987.

[32] W. W.-G. Yeh, "Reservoir management and operations models: a state-of-the-art review," *Water Resources Research*, vol. 21, no. 12, pp. 1797–1818, 1985.

[33] J. D. C. Little, "The use of storage water in a hydroelectric system," *Journal of the Operations Research Society of America*, vol. 3, no. 2, pp. 187–197, 1955.

[34] R. A. Wurbs, "Reservoir-system simulation and optimization models," *Journal of Water Resources Planning and Management*, vol. 119, no. 4, pp. 455–472, 1993.

[35] J. W. Labadie, "Optimal operation of multireservoir systems: state-of-the-art review," *Journal of Water Resources Planning and Management*, vol. 130, no. 2, pp. 93–111, 2004.

[36] S.-M. Choong and A. El-Shafie, "State-of-the-art for modelling reservoir inflows and management optimization," *Water Resources Management*, vol. 29, no. 4, pp. 1267–1282, 2015.

[37] A. B. Celeste and M. Billib, "Evaluation of stochastic reservoir operation optimization models," *Advances in Water Resources*, vol. 32, no. 9, pp. 1429–1443, 2009.

[38] F.-J. Chang, L. Chen, and L.-C. Chang, "Optimizing the reservoir operating rule curves by genetic algorithms," *Hydrological Processes*, vol. 19, no. 11, pp. 2277–2289, 2005.

[39] L. Chen, J. McPhee, and W. W.-G. Yeh, "A diversified multiobjective GA for optimizing reservoir rule curves," *Advances in Water Resources*, vol. 30, no. 5, pp. 1082–1093, 2007.

[40] L. Le Ngo, H. Madsen, and D. Rosbjerg, "Simulation and optimisation modelling approach for operation of the hoa binh reservoir, Vietnam," *Journal of Hydrology*, vol. 336, no. 3-4, pp. 269–281, 2007.

[41] W. Suiadee and T. Tingsanchali, "A combined simulation-genetic algorithm optimization model for optimal rule curves of a reservoir: a case study of the nam oon irrigation project, Thailand," *Hydrological Processes*, vol. 21, no. 23, pp. 3211–3225, 2007.

[42] J. R. Lund and I. Ferreira, "Operating rule optimization for missouri river reservoir system," *Journal of Water Resources Planning and Management*, vol. 122, no. 4, pp. 287–295, 1996.

[43] G. K. Young, "Finding reservoir operating rules," *Journal of the Hydraulics Division*, vol. 93, no. 6, pp. 297–322, 1967.

[44] M. I. Hejazi, X. Cai, and B. L. Ruddell, "The role of hydrologic information in reservoir operation—learning from historical releases," *Advances in Water Resources*, vol. 31, no. 12, pp. 1636–1650, 2008.

[45] Z. Yang, P. Liu, L. Cheng, H. Wang, B. Ming, and W. Gong, "Deriving operating rules for a large-scale hydro-photovoltaic power system using implicit stochastic optimization," *Journal of Cleaner Production*, vol. 195, pp. 562–572, 2018.

[46] M. Karamouz, M. H. Houck, and J. W. Delleur, "Optimization and simulation of multiple reservoir systems," *Journal of Water Resources Planning and Management*, vol. 118, no. 1, pp. 71–81, 1992.

[47] I. Nalbantis and D. Koutsoyiannis, "A parametric rule for planning and management of multiple-reservoir systems," *Water Resources Research*, vol. 33, no. 9, pp. 2165–2177, 1997.

[48] P. Liu, S. Guo, X. Xu, and J. Chen, "Derivation of aggregation-based joint operating rule curves for cascade hydropower reservoirs," *Water Resources Management*, vol. 25, no. 13, pp. 3177–3200, 2011.

[49] A. Cancelliere, G. Giuliano, A. Ancarani, and G. Rossi, "A neural networks approach for deriving irrigation reservoir

operating rules," *Water Resources Management*, vol. 16, no. 1, pp. 71–88, 2002.

[50] A. R. S. Kumar, M. K. Goyal, C. S. P. Ojha, R. D. Singh, P. K. Swamee, and R. K. Nema, "Application of ANN, fuzzy logic and decision tree algorithms for the development of reservoir operating rules," *Water Resources Management*, vol. 27, no. 3, pp. 911–925, 2013.

[51] V. Chandramouli and H. Raman, "Multireservoir modeling with dynamic programming and neural networks," *Journal of Water Resources Planning and Management*, vol. 127, no. 2, pp. 89–98, 2001.

[52] M. Sangiorgio and G. Guariso, "NN-based implicit stochastic optimization of multi-reservoir systems management," *Water*, vol. 10, no. 3, p. 303, 2018.

[53] P. Chaves and T. Kojiri, "Deriving reservoir operational strategies considering water quantity and quality objectives by stochastic fuzzy neural networks," *Advances in Water Resources*, vol. 30, no. 5, pp. 1329–1341, 2007.

[54] S. J. Mousavi, K. Ponnambalam, and F. Karray, "Inferring operating rules for reservoir operations using fuzzy regression and ANFIS," *Fuzzy Sets and Systems*, vol. 158, no. 10, pp. 1064–1082, 2007.

[55] C.-C. Wei and N.-S. Hsu, "Derived operating rules for a reservoir operation system: comparison of decision trees, neural decision trees and fuzzy decision trees," *Water Resources Research*, vol. 44, no. 2, Article ID W02428, 2008.

[56] L. Li, P. Liu, D. E. Rheinheimer, C. Deng, and Y. Zhou, "Identifying explicit formulation of operating rules for multi-reservoir systems using genetic programming," *Water Resources Management*, vol. 28, no. 6, pp. 1545–1565, 2014.

[57] C.-M. Ji, T. Zhou, and H.-T. Huang, "Operating rules derivation of jinsha reservoirs system with parameter calibrated support vector regression," *Water Resources Management*, vol. 28, no. 9, pp. 2435–2451, 2014.

[58] A. B. Alaya, "Optimization of nebhana reservoir water allocation by stochastic dynamic programming," *Water Resources Management*, vol. 17, no. 4, pp. 259–272, 2003.

[59] P. P. Mujumdar and B. Nirmala, "A bayesian stochastic optimization model for a multi-reservoir hydropower system," *Water Resources Management*, vol. 21, no. 9, pp. 1465–1485, 2007.

[60] V. Jothiprakash and G. Shanthi, "Comparison of policies derived from stochastic dynamic programming and genetic algorithm models," *Water Resources Management*, vol. 23, no. 8, pp. 1563–1580, 2009.

[61] R. Yun, V. P. Singh, and Z. Dong, "Long-term stochastic reservoir operation using a noisy genetic algorithm," *Water Resources Management*, vol. 24, no. 12, pp. 3159–3172, 2010.

[62] D. Etkin, P. Kirshen, D. Watkins et al., "Stochastic programming for improved multiuse reservoir operation in Burkina Faso, west Africa," *Journal of Water Resources Planning and Management*, vol. 141, no. 3, Article ID 04014056, 2015.

[63] M. E. El-Hawary and G. S. Christensen, *Optimal Economic Operation of Electric Power Systems*, Academic Press, Cambridge, MA, USA, 1979.

[64] G. W. Tauxe, R. R. Inman, and D. M. Mades, "Multiple objectives in reservoir operation," *Journal of the Water Resources Planning and Management Division*, vol. 106, no. 1, pp. 225–238, 1980.

[65] C. Lyra and L. R. M. Ferreira, "A multiobjective approach to the short-term scheduling of a hydroelectric power system," *IEEE Transactions on Power Systems*, vol. 10, no. 4, pp. 1750–1755, 1995.

[66] X. Zhang, X. Yu, and H. Qin, "Optimal operation of multi-reservoir hydropower systems using enhanced comprehensive learning particle swarm optimization," *Journal of Hydro-Environment Research*, vol. 10, pp. 50–63, 2016.

[67] M. S. Hossain and A. El-Shafie, "Intelligent systems in optimizing reservoir operation policy: a review," *Water Resources Management*, vol. 27, no. 9, pp. 3387–3407, 2013.

[68] D. A. Augusto and H. J. C. Barbosa, "Accelerated parallel genetic programming tree evaluation with opencl," *Journal of Parallel and Distributed Computing*, vol. 73, no. 1, pp. 86–100, 2013.

[69] A.-K. C. Ahamed and F. Magoules, "Conjugate gradient method with graphics processing unit acceleration: Cuda vs opencl," *Advances in Engineering Software*, vol. 111, pp. 32–42, 2017.

[70] S. K. Park and K. W. Miller, "Random number generators: good ones are hard to find," *Communications of the ACM*, vol. 31, no. 10, pp. 1192–1201, 1988.

[71] M. Taherkhani and R. Safabakhsh, "A novel stability-based adaptive inertia weight for particle swarm optimization," *Applied Soft Computing*, vol. 38, pp. 281–295, 2016.

[72] X. Xia, J. Liu, and Z. Hu, "An improved particle swarm optimizer based on tabu detecting and local learning strategy in a shrunk search space," *Applied Soft Computing*, vol. 23, pp. 76–90, 2014.

[73] H. Wang, W. Wang, X. Zhou et al., "Firefly algorithm with neighborhood attraction," *Information Sciences*, vol. 382-383, pp. 374–387, 2017.

[74] H. Wang, X. Zhou, H. Sun et al., "Firefly algorithm with adaptive control parameters," *Soft Computing*, vol. 21, no. 17, pp. 5091–5102, 2017.