applied sciences

MDPI

Article

On GPU Implementation of the Island Model Genetic Algorithm for Solving the Unequal Area Facility Layout Problem

Xue Sun [1,2], Lien-Fu Lai [3], Ping Chou [4], Liang-Rui Chen [2] and Chao-Chin Wu [3,*]

1 College of Urban Rail Transit and Logistics, Beijing Union University, Beijing 100101, China; zdhtsunxue@buu.edu.cn
2 Department of Electrical Engineering, National Changhua University of Education, Changhua 50007, Taiwan; lrchen@cc.ncue.edu.tw
3 Department of Computer Science and Information Engineering, National Changhua University of Education, Changhua 50007, Taiwan; lflai@cc.ncue.edu.tw
4 Department of Management Information Systems, National Chengchi University, Taipei 11605, Taiwan; littlesixdot@gmail.com
* Correspondence: ccwu@cc.ncue.edu.tw; Tel.: +886-4-7232105 (ext. 8431)

Received: 12 August 2018; Accepted: 7 September 2018; Published: 10 September 2018

Abstract: Facility layout problem (FLP) is one of the hottest research areas in industrial engineering. A good facility layout can achieve efficient production management, improve production efficiency, and create high economic values. Because FLP is an NP-hard problem, meaning it is impossible to find the optimal solution when problem becomes sufficiently large, various evolutionary algorithms (EAs) have been proposed to find a sub-optimal solution within a reasonable time interval. Recently, a genetic algorithm (GA) was proposed for unequal area FLP (UA-FLP), where the areas of facilities are not identical. More precisely, the GA is an island model based, which is called IMGA. Since EAs are still very time consuming, many efforts have been devoted to how to parallelize various EAs including IMGA. In recent work, Steffen and Dietmar proposed how to parallelize island models of EAs. However, their parallelization approaches are preliminary because they focused mainly on comparing the performances between different parallel architectures. In addition, they used one mathematical function to model the problem. To further investigate on how to parallelize the IMGA by GPU, in this paper we propose multiple parallel algorithms, for each individual step in the IMGA when solving the industrial engineering problem, UA-FLP, and conduct experiments to compare their performances. After integrating better algorithms for all steps into the IMGA, our GPU implementation outperforms the CPU counterpart and the best speedup can be as high as 84.

Keywords: unequal area facility layout problem; parallel computing; island model; genetic algorithm; GPU

1. Introduction

In manufacturing industries, facility layout problems (FLP) are one of the most important issues among the various aspects of manufacturing system management. FLPs have a significant impact upon manufacturing costs, lead times, work processes and productivity [1]. The suitable placement of facilities contributes to reduction of the total operating costs by 20% to 50% [2]. Based on shapes and dimensions, FLPs can be divided into different categories. Among them, unequal area FLP (UA-FLP) has been attracting attention of many researchers because it has been applied to many fields such as industrial facility design, warehouse organization, school facility layout, and VLSI element placement [3,4].

In most studies, the main goal for UA-FLP is to obtain the minimal operating costs [5]. However, UA-FLP has been proven to be an NP-hard combinatorial optimization problem. That is, when the number of facilities is increased, the computational time is exponentially increased [6]. As a sequence, during the last decades more and more research has proposed various metaheuristic approaches to solve UA-FLP by finding the approximate optimal solution in finite time [7–12]. Among them, the most popular and widely used research is genetic algorithm (GA).

GA, proposed by Holland [13] and DeJong [14], is based on Darwin's principles of natural selection and divergence [15]. GA is a global search algorithm, which has typical applications in different fields to solve optimization problems [16–20]. For instance, recently GA has been proven for its effectiveness in enhancing the opportunity to achieve a global optimal solution without falling into the local optimal solution to UA-FLP [11,12]. However, approaches based on standard GAs usually have premature convergence, which means a population for an optimization problem converged too early to get optimal solution. Moreover, all the individuals in a population should be selected, evaluated, crossed, and mutated in each generation, which requires a long CPU execution time [21].

Many models of parallel GAs have been proposed to shorten the computation time in the literature. Two categories of parallel GAs are island model and global parallelization [22]. In recent work, Steffen Limmer et al. [22] compared four parallel architectures including multi-core CPUs, clusters, grids, and graphic process units (GPUs) for the execution of the two models of the genetic algorithms. In this study, we pay close attention to how to use GPU to parallelize island model of genetic algorithm (IMGA).

Nowadays, modern GPUs have evolved into very powerful and flexible processors. Especially after CUDA (compute unified device architecture) platform were distributed, developing highly parallel GPU applications becomes much easier. GPUs are very well to address general problems that are suitable for data-parallel computations, including GA [23–25]. The IMGA, a parallel genetic algorithm model, can fully explore the computing power by either coarse or fine-grained parallelisms. It is worthwhile to study how to parallelize IMGA on GPUs to solve real-world optimization problems. Nevertheless, only few works have discussed this issue before. N. Melab and E. G. Talbi [26] proposed three different general schemes for building efficient island models for GA on GPU. Their experiments indicated that GPU computing can not only speed up the search process, but also exploit parallelism to improve the quality of the obtained solutions. The authors strongly believed that the schemes of the fully distributed island model on GPU could be easily extended to large scale optimization problems. Steffen and Dietmar [22] discussed the island model and global parallelization for evolutionary algorithm in detail. The methods of IMGA on GPU are based on the work reported in the work [26], and a common Weierstrass function is used to compare the performances between GPU and CPU. However, their parallelization approaches are preliminary because they focused mainly on comparing the performances between different parallel architectures. In addition, they used one mathematical function to model the problem.

In this paper, we further investigate how to parallelize the IMGA by GPU. We propose multiple parallel algorithms, for each individual step in the IMGA when solving UA-FLP, and conduct experiments to compare their performances. To the best of our knowledge, no previous work has considered using GPU parallelization to solve UA-FLP. Through comparing the different parallel schemes at each step in the IMGA, the better ones were integrated as an entire program of IMGA. According to our experimental results of IMGA, our GPU implementation outperforms the CPU counterpart and the best speedup can be as high as 84.

The paper is organized as follows. Section 2 introduces the related work, including introduction to CUDA GPUs, IMGAs, and UA-FLPs. Section 3 presents our multiple parallel strategies for the main steps of IMGA on a GPU. Through theoretical analysis and experimental comparison of these parallel methods for each step, the better ones are integrated into our IMGA. Moreover, the overall performance of the algorithm on the GPU and CPU is compared through experiments. Finally, conclusions are given in Section 4.

2. Related Work

In industry, data processing is one of the most important technologies—especially when the paradigm of "manufacturing as an ecosystem" has emerged. Modern information and communication technologies such as big data analytics and cloud computing [27,28] can provide useful insight to the industry to increase productivity, quality, and agility benefits, which have significance competitive value. For a long time, the industry has been looking for efficient algorithms to solve UA-FLPs, which are typical NP-hard problems. That is, their solution space is characterized by rapid expansion as the scale of the problem increases. The use of mathematical programming methods to solve large-scale and complex problems is difficult to achieve satisfactory results. People are more inclined to seek heuristic algorithms, for example GA, that find acceptable approximate solutions in a limited time. However, the standard GAs also require a long CPU execution time, so many parallel architectures are proposed. The parallel architectures, such as multi-core CPU, grid, clusters, and GPUs are widely used in many fields [27–29]. Steffen Limmer et al. [22] mainly compare these four architectures for IMGA to solve a mathematic function. In this work, we emphasize using GPU to solve IMGA for UA-FLP. In this section, the related work is surveyed, including CUDA GPUs, IMGAs, and UA-FLPs.

2.1. CUDA GPU

During the last few years, the capability of GPU is growing much faster than that of CPU's because of greatly increasing hardware requirement for modern computer games. The GPU is also rapidly and widely used for various scientific computations in addition to graphic display [30], such as fluid dynamics [31], biophysics [32], molecular dynamics [33], and IoT sensing [34]. GPUs can provide huge performance improvement than a single CPU core for many applications. For instance, the work in the CFD field [31] has shown speed-up values of a single GPU against a single core CPU larger than 100. 'General-purpose computing on graphics processing unit' (GPGPU) was thus coined [35].

On a GPU-chip, there are multiple cores called streaming-multiprocessors (SMs), and each SM has several streaming processors (SPs). Each SP of an SM executes the same instruction on different data during each cycle. Since 2007, NIVDIA has distributed CUDA as a platform to develop on GPU. Along with the introduction of CUDA platform, a rapid increase of scientists paid more attention to GPU because CUDA makes programmers easier work. A CUDA program can be executed on the host (CPU) and on a device (GPU). Sequential codes run on the CPU, while the parallel codes are offloaded to the GPU, called a 'kernel'. If a kernel is invoked, blocks with the grid are distributed to the SM. The kernel can launch up a large number of threads to exploit data parallelism. The warps containing 32 threads in each of the blocks exhibit single instruction multiple data (SIMD) execution. All threads within a warp must execute the same instructions at any given time. Data-dependent branch causes different threads in the same warp to follow different paths, known as branch divergence [36]. Branch divergence has a significant impact on the performance of GPU. CUDA GPU owns many memories—including global memory, shared memory, constant memory, local memory, and registers—to achieve high execution speed in their kernel. Different types of memory are differing in sizes, access scopes, access times, and whether it is read-only or cached. In brief, GPUs are now becoming the most suitable processor for implementing algorithms with simple and large amount of computations [30].

2.2. IMGA

IMGA is one of the most promising variant among the parallelized models of GA. The model uses the advantage of parallel computing by dividing population of a large size into smaller partitions, distributing them into different islands to perform simultaneous random searches, and exchanging the individuals through a migration operator. The IMGA model is possible to explore different regions of the search space, so as to improve issues of premature convergence or convergence to local optimal solutions in the standard GA using a single large population. In the latest literature,

an IMGA for UA-FLP was proposed by J. M. Palomo-Romero et al. [21]. This is the first time to propose a parallel GA based on the island model to solve UA-FLP using the flexible bay structure (FBS) formulation. In their proposed approach, each island is assigned with a part of population for local evolution, and several chromosomes of high quality will migrate from one island to another after a few generations. The experiments were conducted on a total of 26 benchmarks of UA-FLP, and the results were compared with the best results found in the literature. The method can improve search quality, obtain better solutions, provide greater population diversity, and reduce the number of evaluations required to find good solutions. Moreover, the execution times of the algorithm were similar to, or lower than, the execution times of the previous approaches in all cases. Nevertheless, the IMGA method still requires long execution time, which is why the authors pointed an interesting future work in their paper, i.e., parallelizing IMGA. Parallelizing IMGA on GPU is helpful to reduce the execution time in searching for good solutions especially for large-scale problems and further solving the issues of local optimal solutions. In our study, we propose multiple parallel algorithms to implement each step of IMGA and compare the performance ratios between them. The better methods for all steps are integrated into our parallel optimized model for UA-FLP on GPU, which are described in the following section.

2.3. UA-FLP

The UA-FLP was originally formulated by Armour and Buffa [37]. It considered on the assumptions that there is a given rectangular region, and there are a set of facilities needed to be placed into the region. The fixed dimensions of the rectangular region are $W(width) \times H(height)$, and each of the rectangular facilities occupies a specified area (B_i). The constraints of the problem include the sum of the facility areas must be less than or equal to the fixed rectangular region (see Equation (1)) and the facilities cannot overlap. The objective function, or the fitness function in our study, is on basis of total material handling cost (MFC) between facilities [38], which is presented in Equation (2).

$$\sum_{i=1}^{p} B_i \leq W \times H \tag{1}$$

$$\min MFC = \sum_{i=1}^{p} \sum_{j=1, i \neq j}^{p} f(i,j)d(i,j) + (D_{\text{inf}})^k (V_{feas} - V_{all}) \;\; i,j = 1,2,\ldots,p, \tag{2}$$

where:

p = the number of facilities;
$f(i,j)$ = the total flow of moving materials between facility i and j, where $i,j = 1,2,\ldots,p$;
$d(i,j)$ = the weighted rectilinear distance between facilities i and j, where $i,j = 1,2,\ldots,p$;
D_{inf} = the number of infeasible departments;
V_{feas} = the best feasible objective function value yet found;
V_{all} = the best overall objective function value yet found;
k = the parameter for adjusting the "severity", which was set to $k = 3$ [39].

The calculation of distance between facilities can be Euclidean (Equation (3)) or rectilinear (Equation (4)), where the point defined by x and y is the center of the facility

$$d(i,j) = \sqrt{(x_j - x_i)^2 + (y_i - y_j)^2} \tag{3}$$

$$d(i,j) = |x_i - x_j| + |y_i - y_j| \tag{4}$$

The FBS in our UA-FLP consists of two parts: the facility sequence codes and the bay division codes. The former encodes the order of p facilities represented by integer numbers which will be placed into the fixed rectangular area. The order of the facilities, named 'facility sequence', is organized

bay by bay from top to bottom and from left to right. The latter encodes the bay divisions which have $(p-1)$ binary elements in the area, each facility is associated with one element except the first one. A value of 1 indicates that the corresponding facility is the last facility in the present bay, while 0 means the facility and its previous one are in the same bay. We take an example to detail the encoding method of FSB.

Assume there are seven facilities in a UA-FLP. The facility layout problem is to minimize the overall cost by determining how many bays are required, which facilities are assigned to which bays, what kind of ordering is adopted for placing the facilities in each bay. For instance, one solution is illustrated in Figure 1.

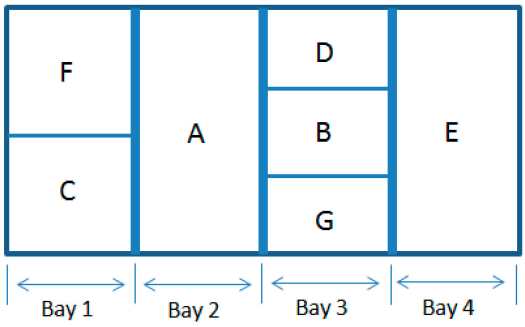

Figure 1. An example of unequal sized facility layout.

There are four bays in this example: Bay 1, Bay 2, Bay 3, and Bay 4. Each bay contains one or more facilities. For instance, Bay 1 contains two facilities, F and C, while Bay 2 contains only one facility, A. To represent the layout shown in Figure 1 by using FSB encoding, two arrays are required, as shown in Figure 2. The first array, facility sequence, describes the ordering of the facilities from top to bottom and from left to right in the layout. The second array, bay divisions, indicates whether the corresponding facility is the last one in one bay or not. Since the top left facility is F, it is the first one placed into the two arrays, as shown the first step in Figure 2. Moreover, because it is not the last one in the first bay, the corresponding element is set to 0. That is, the first element in the array of bay divisions is set to 0. Next, the second facility, C, is inserted into the arrays because it is in Bay 1 and below Facility F, as shown in Step 2 in Figure 2. Since it is the last one in Bay 1, its value in the second element in the array of bay division is set to 1. At Step 3, we move to Facility A in Bay 2. Because Facility A is also the last one in Bay 2, the third element in Bay Divisions is set to 1. Totally, seven steps are needed for processing seven facilities. Note that the seventh element in the bay divisions array is omitted because Facility E is the last one in the facility sequence.

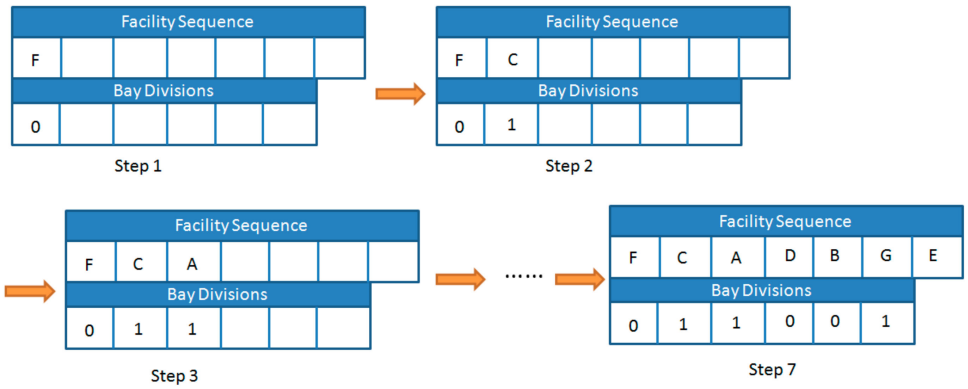

Figure 2. Steps of FBS encoding for the facility layout shown in Figure 1.

3. Our Parallel Strategies

In this section, our parallel strategies for the main steps of IMGA are proposed in detail. Section 3.1 illustrates the overall structure of our parallel IMGA on GPU, and introduces our experimental environment. Sections 3.2–3.5 explain the multiple parallel strategies for initialization, selection, mutation, and migration of IMGA on GPU, respectively, and compare the performance ratio between the multiple parallel versions. Section 3.6 is the comparison of the performance executed on GPU with that on CPU counterpart after integrating the better parallel strategies.

3.1. Parallel IMGA on GPU

The sequential IMGA consists of the following processes [21]. (1) n individuals as the initial population is randomly generated and distributed into N islands randomly. (2) All newly-generated individuals should be evaluated fitness values by the objective function. The processes of next procedures from (3) to (7) are performed sequentially on each island, which repeat c times to produce the best individuals. (3) The best individuals are selected for recombination to the subpopulation with selection operators. (4) Two individuals are recombined randomly using crossover operators. (5) Mutation operators should be conducted for avoiding local solutions for maintaining genetic diversity. (6) The fitness values of modified individuals are updated. The population for the next generation is generated and takes place of the parents. (7) Individuals between the islands are migrated after g generations.

Figure 3 illustrates the structure of our parallel IMGA on GPU. The main implementation structure is shown in Figure 3a. There are N islands organized as ring topology, which have n/N chromosomes on each of them. The entire algorithm is executed on GPU with one block per island. The total population of individuals is stored in global memory which is the largest in terms of size. For each block, a group of threads execute standard GA in parallel. Figure 3b gives an illustration of one particular island on GPU. Threads on each block are responsible for executing the procedures of IMGA including initialization, evaluation, selection, crossover, mutation, replacement, and migration operators. Regarding the migration, communications are carried out through the global memory which stores global population, resulting in each island being able to communicate with each other according to our ring topology.

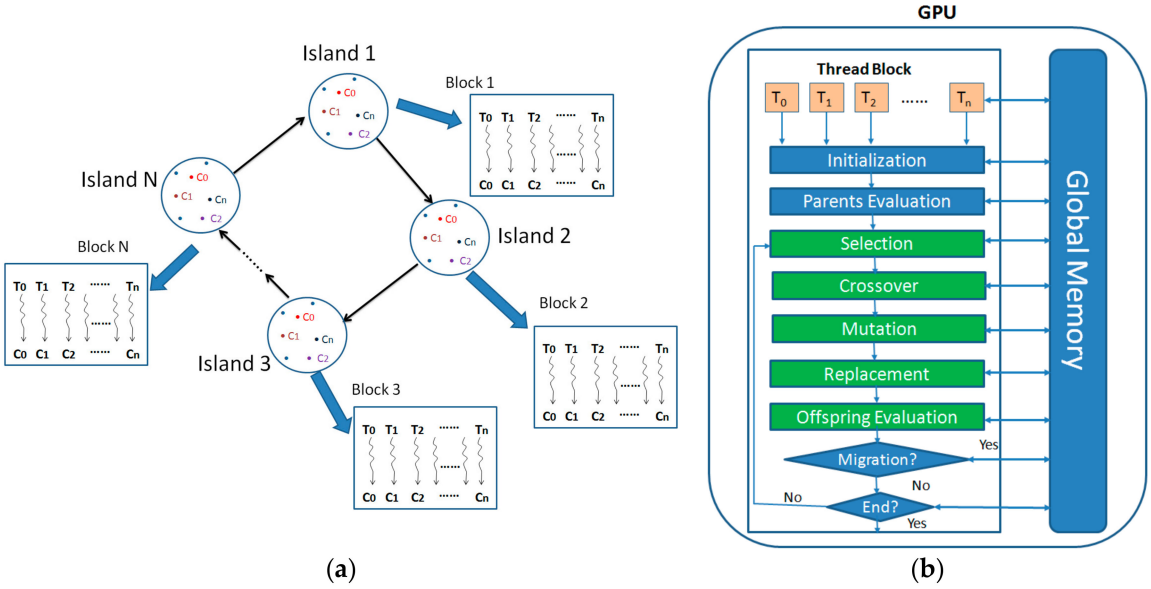

Figure 3. The structure of our parallel IMGA on GPU. (**a**) Description of the main implementation structure. (**b**) Implementation of one particular island on GPU.

In our CUDA GPU programming, one-dimensional array of blocks and threads is used in the application of UA-FLP. The program starts on the host, i.e., a CPU core. Parallel computations are conducted on the device—i.e., a GPU—where the kernel functions are used to do the parallel executions. The main kernel functions in our program are generation of facility sequence, generation of facility bay divisions, fitness calculation, selection, crossover, mutation, and migration. In the following subsections, we select the main operations to illustrate our GPU solutions to parallelize IMGA for UA-FLP. In the previous work [22], for each step of IMGA, only one method is used to implement the parallelization. In our implementation, multiple parallelized schemes are proposed to improve the main operations. After comparing these schemes for each step, the methods with better performances are determined. In the following subsections, we explain our multiple implementations for the main steps in detail.

We adopt NVIDIA GeForce GTX 980 [40] to evaluate our GPU version of IMGA. The workstation mainly consists of Intel Xeon CPU E5-2609 v3 with 16GB memory and GXT980 with 4GB memory. Detailed configurations are shown in Table 1. Our GPU and CPU versions are written in C, and the operating system installed is Linux and its version is Ubuntu 14.04 64-bit. We use CUDA version 7.0 to implement the GPU algorithms. In the experiments, for comparison with the performance of the different versions, we adopt performance ratio between the multiple methods for measurement.

Table 1. Configuration of workstation.

Intel Xeon CPU E5-2609 v3		**GTX 980**	
Number of Cores	6	Number of GPUs	1
Number of Threads	6	Thread Processors	2048
Clock Speed	1.9 GHZ	Clock Speed	1127 MHZ
Memory Size	16 GB	Memory Size	4 GB
Memory Type	DDR4	Memory Type	GDDR5

Our proposed algorithms are tested using 26 well-known problem sets taken from the literature [21] as shown in Table 2. For testing larger scale facilities, we have added 8 problem data sets. Table 3 shows the information about our added problem data sets. In Tables 2 and 3, number of facilities, facility size ($width \times length$), shape constraint (α is the maximum aspect radio constraint, and $l_{\min}$ is the minimum side length constraint, and distance measure (Euclidean or rectilinear) are illustrated in detail. Note that the number in the suffix of each data set name conventionally indicates how many the facilities the data set contains. For instance, VC10E-a has 10 facilities and F40 has 40 facilities.

Some parameters are needed for our IMGA such as total population size (n), number of islands (N), cycle generation (c), migration rate (m), migration frequency (g), crossover probability (p_c), and mutation probability (p_m). We also tested empirically to determine the better fit parameters when developing our programs. In the following experiments, the settings of important parameters are listed in Table 4, and we have executed five times for each data set to calculate its average execution time and solution quality.

In the following subsections, all the tests are based on 15 islands and 500 individuals per island, and we choose the typical examples of different number of facilities in the benchmark to conduct the experiments. The kernel configuration is shown in Table 5, which indicates the number of blocks (B) and the number of threads per block (T) used for each kernel function.

Table 2. The 26 well-known problem data sets.

No.	Problem Data Set	Number of Facilities	Facility Size		Common Shape Constraint	Distance Measure
			Width	Length		
1	O7	7	8.54	13.00	$\alpha = 4$	Rectilinear
2	O8	8	11.31	13.00	$\alpha = 4$	Rectilinear
3	O9	9	12.00	13.00	$\alpha = 4$	Rectilinear
4	VC10E-a	10	25.00	51.00	$\alpha = 5$	Euclidean
5	VC10E-s	10	25.00	51.00	$lmin = 5$	Euclidean
6	VC10R-a	10	25.00	51.00	$\alpha = 5$	Rectilinear
7	VC10R-s	10	25.00	51.00	$lmin = 5$	Rectilinear
8	Ba12	12	6.00	10.00	$lmin = 1$	Rectilinear
9	Ba12TS	12	6.00	10.00	$lmin = 1$	Rectilinear
10	MB12	12	6.00	8.00	$\alpha = 4$	Rectilinear
11	Ba14	14	7.00	9.00	$lmin = 1$	Rectilinear
12	Ba14TS	14	7.00	9.00	$lmin = 1$	Rectilinear
13	AB20-ar1000	20	2.00	3.00	$\alpha = 1000$	Rectilinear
14	AB20-ar50	20	2.00	3.00	$\alpha = 50$	Rectilinear
15	AB20-ar25	20	2.00	3.00	$\alpha = 25$	Rectilinear
16	AB20-ar15	20	2.00	3.00	$\alpha = 15$	Rectilinear
17	AB20-ar10	20	2.00	3.00	$\alpha = 10$	Rectilinear
18	AB20-ar7	20	2.00	3.00	$\alpha = 7$	Rectilinear
19	AB20-ar5	20	2.00	3.00	$\alpha = 5$	Rectilinear
20	AB20-ar4	20	2.00	3.00	$\alpha = 4$	Rectilinear
21	AB20-ar3	20	2.00	3.00	$\alpha = 3$	Rectilinear
22	AB20-ar2	20	2.00	3.00	$\alpha = 2$	Rectilinear
23	AB20-ar175	20	2.00	3.00	$\alpha = 1.75$	Rectilinear
24	AB20-ar170667	20	2.00	3.00	$\alpha = 1.70667$	Rectilinear
25	SC30	30	12.00	15.00	$\alpha = 5$	Rectilinear
26	SC35	35	15.00	16.00	$\alpha = 4$	Rectilinear

Table 3. Our added data of test problem sets.

No.	Problem Data Set	Number of Facilities	Facility Size		Common Shape Constraint	Distance Measure
			Width	Length		
1	F40	40	35.00	35.00	$\alpha = 1000$	Rectilinear
2	F45	45	35.00	35.00	$\alpha = 1000$	Rectilinear
3	F50	50	35.00	35.00	$\alpha = 1000$	Rectilinear
4	F55	55	45.00	45.00	$\alpha = 1000$	Rectilinear
5	F60	60	45.00	45.00	$\alpha = 1000$	Rectilinear
6	F65	65	45.00	45.00	$\alpha = 1000$	Rectilinear
7	F70	70	45.00	45.00	$\alpha = 1000$	Rectilinear
8	F75	75	45.00	45.00	$\alpha = 1000$	Rectilinear

Table 4. Settings of the parameters in our program.

Parameter	Value
Total population size per island (n/N)	500
Number of islands (N)	15
Cycle generations (c)	70
Migration rate (m)	5
Migration frequency (g)	15
Crossover probability (p_c)	0.7
Mutation probability (p_m)	0.01

Table 5. Configuration of each kernel function.

Kernel Function	Number of Blocks (B)	Number of Threads per Block (T)
Parallel facility sequences generation	15	500
Parallel bay divisions generation	15	500
Parallel roulette wheel selection	15	500
Parallel mutation operator	15	500
Parallel migration operator	15	5*β [1]

[1] The β is equal to 1 when one thread is used to migrate one chromosome; and the β is equal to the number of genes per chromosome when one thread is used to migrate one gene.

3.2. Parallel Initialization on GPU

In data initialization, random numbers are used to generate the initial content of each individual. In our case of UA-FLP, because the FBS is used to encode our facility layout, initialization of facilities consists of two parts: generations of facility sequence and bay divisions, which will be detailed below.

3.2.1. Facility Sequence Generation

To generate facility sequences randomly, the conventional method (INI_FS1) is usually adopted, i.e., if there are p facilities, p random numbers are generated and any two numbers cannot be identical. An example to illustrate INI_FS1 is shown in Figure 4. Assume there are seven facilities, and they are represented by letters, A to G, respectively. Actually, each letter is associated with a code, e.g., 1 to 7 for A to G. Initially, we need a facility sequence to generate different facility sequences from it, which is called the seed facility sequence. The seed facility sequence is set according to alphabetical order, as shown on the left side of Figure 4. To generate a new individual of the facility sequence, Step 1 is to produce one random integer number, 5 in this example. Then at Step 2, the second random integer number, 3, is produced. In order to avoid generating two identical facilities, we have to compare the newly generated facility with all previous ones at every step. Therefore, at Step 2, the second number, 3, is compared with the first number, 5, and we conclude that 3 can be kept. Next, at Step 3 the number 5 is produced, however, it is identical to the first number in the sequence. Therefore, the newly generated facility with number 5 should be abandoned and regeneration of the third number is required. At Step 4, the code number 4 is randomly produced, which can be accepted through comparing it with the preceding numbers. Repetition of random number generation and duplication avoidance like above steps, finally a new facility sequence is generated, as shown on the right-hand side of Figure 4. In this method, a large number of comparisons are required, making the complexity of the algorithm equal to $O(p^2)$. Therefore, our improved method (INI_FS2) is proposed below to eliminate comparisons and also to avoid regeneration of random numbers for the same array element.

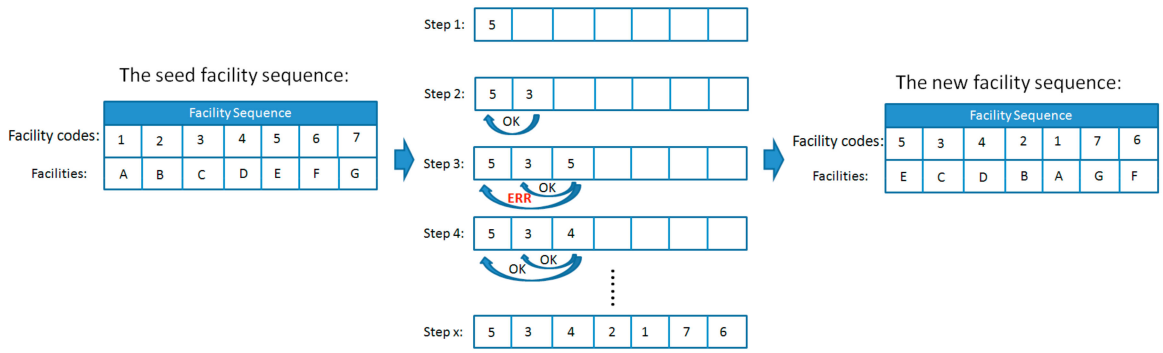

Figure 4. An example of the INI_FS1 method.

The main idea of INI_FS2 method is to get a new facility sequence by performing a swap operation several times on the seed facility sequence to fully randomizing the facility positions. If there are

p facilities, p swaps are performed. How the p swaps are performed is based on an array, called 'swap position', which contains p random numbers and the value of each random number is between 0 and $(p-1)$. It is not required that any two of the p random numbers are not identical, which is crucial to eliminate comparisons of duplicates. The random numbers stored in the array of swap position will be extracted from the head to the end, one by one, to perform a swap on the seed facility sequence. At Step i, we exchange two positions in the seed facility sequence, one is the position $(p-1-i)$, another is the position specified by the random number stored in the $(i-1)$-th element in the array of swap position. An example is shown in Figure 5 to illustrate the method of INI_FS2. Firstly, we generate seven random numbers without checking duplicate and store them in the array of swap position, as shown on the left-hand side of Figure 5. At Step 1, because the first random number in the array of swap position is 5, we exchange the two elements on Positions 6 and 5 in the array of facilities. At Step 2, because the second random number in swap position is 3, Positions 3 and 5 in the facilities array are exchanged. Similarly, Positions 2 and 5 are swapped at Step 3. After seven steps, we get a new facility sequence, as shown on the right hand side of Figure 5. The complexity of the INI_FS2 method is O(p) since no comparisons are required.

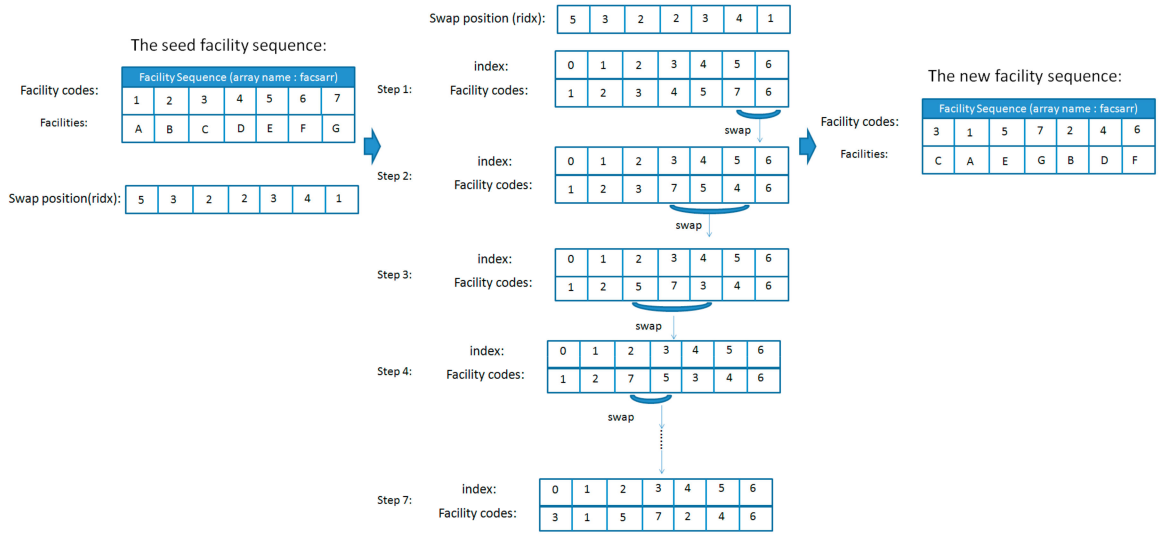

Figure 5. An example of the INI_FS2 method.

The kernel Algorithm 1 shows the pseudocode for INI_FS2 method to parallelize the generation of facility sequences. We adopt a coarse-grained parallelization method. That is, every thread will produce a new facility sequence based on the seed facility sequence. It is inefficient to adopt fine grained parallelization method, as explained below. If multiple threads generate random number simultaneously to produce a facility sequence, they need to check whether any two random numbers are identical because any facility sequence is actually a permutation of facility IDs in UA-FLP. It incurs high computational cost when multiple parallel threads need to perform comparison, communication, and even synchronization to eliminate duplicates.

In the Kernel Algorithm 1, firstly, each thread generates random numbers and stores them in the array of ridx based on its thread id and block id. Next, each thread reads one random number from array ridx and uses it to exchange two positions on the corresponding facility sequence in array facsarr.

Algorithm 1: Parallel Facility Sequences Generation

```
Data:
    blockID: Block id
    threadID: Thread id
    length: The number of facilities
    ridx: Array of Swap Position
    facsarr: Array of the seed facility sequence
Input:
    facsarr;
Result:
    New facility sequences generated randomly
Kernel function:
    __global__ void genF (Array facsarr, Array ridx) {
      1:  Generate ridx[blockID][threadID] with random numbers;
      2:  foreach i ∈{0, . . . ,length−1} do
      3:    pos ← ridx[blockID][threadID][i];
      4:    Swap(facsarr[blockID][threadID][length − i − 1], facsarr[blockID][threadID][pos]);
      5:  end
    }
```

The performance ratio between INI_FS2 and INI_FS1 for facility sequence generation is shown in Figure 6. According to the results, the INI_FS2 is better than the INI_FS1 especially when the number of facilities is increased.

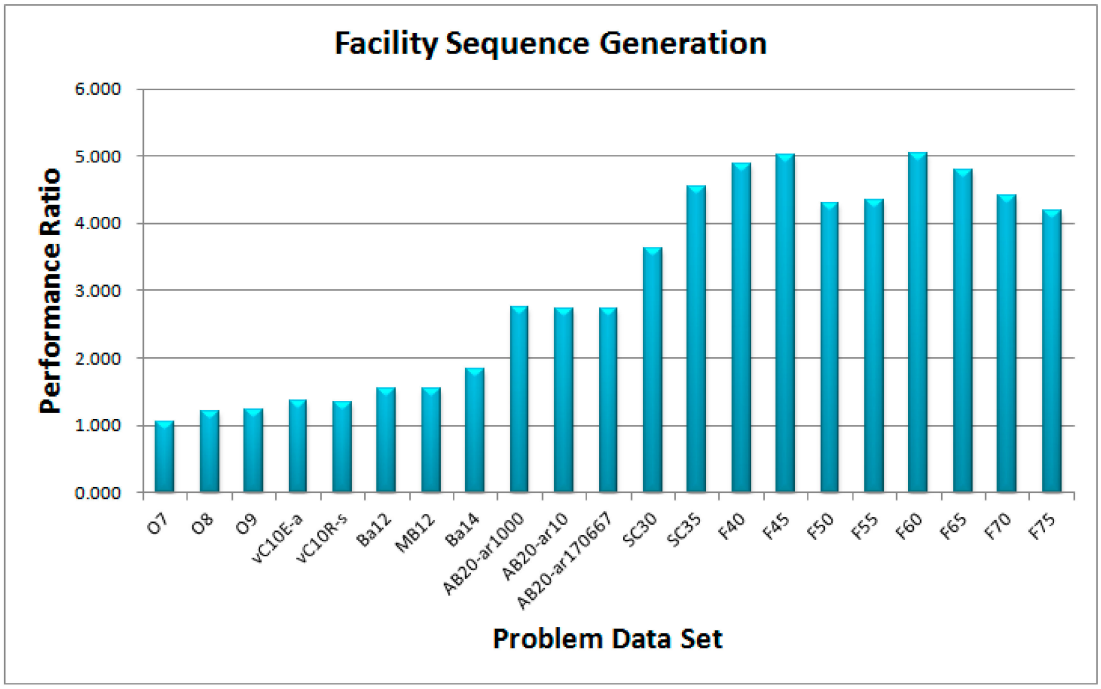

Figure 6. Performance ratio between INI_FS2 and INI_FS1 for facility sequence generation.

3.2.2. Bay Divisions Generation

To generate bay divisions, the conventional method and the improved method are both proposed in this subsection. The conventional method (INI_BD1) is firstly to generate a random integer *randI* for each facility on bay divisions, and then use *randI*%2 to get an integer of value equal to 0 or 1. Consequently, $(p-1)$ random integers are required for p facilities. In INI_BD1, p facilities require $(p-1)$ modulo operations for the representation of bay divisions. If there are n individuals, we need

to perform the modulo operation $(n * (p-1))$ times in total, thus resulting in a high time complexity even in parallel computing. In order to reduce such calculations in parallel, we propose the other method (INI_BD2) to improve the performance of generating bay divisions. The new method uses bitwise operators instead of modulo operators. Assume the value of an integer generated randomly is x $(x \geq 1)$, we calculate x AND 1 to generate a value of either 0 or 1 for the first facility in the bay divisions. Then a one-bit right arithmetic shift is adopted to change the value of x, and the new x is used in next iteration. Repeat the above procedure for $(p-1)$ times, the whole bay divisions are generated. INI_BD2 algorithm is of O(1) time complexity, which is much smaller than O(p) time complexity required by INI_BD1. Moreover, INI_BD2 uses arithmetic shift operations to replace modulo operators in INI_BD1.

$$x\ AND\ 1 = \sum_{i=0}^{\lfloor \log_2(x) \rfloor} 2^i \left(\left\lfloor \frac{x}{2^i} \right\rfloor \bmod 2 \right) \left(\left\lfloor \frac{1}{2^i} \right\rfloor \bmod 2 \right) \tag{5}$$

$$x_{new} = x * 2^b \tag{6}$$

Kernel Algorithm 2 is the pseudocode of the improved method INI_BD2 for facility bay divisions. Each thread is responsible for producing one bay divisions. Firstly, we generate a random integer, and then bitwise operators, instead of modulo operators, are used to obtain a value of either 0 or 1 for every facility except the last one (Lines 2–5).

Algorithm 2: Parallel Bay Divisions Generation

```
Data:
    decimal: Variable for a random integer value
    baydarr: Boolean array containing flexible bay divisions for each individual
Operator:
    '>>': Bitwise operation SHR (shift right)
    '&': Bitwise operation AND
Input:
    baydarr
Result:
    All of facility divisions generated randomly
Kernel function:
__global__ void genB (Array baydarr) {
1:  decimal <- Random()
2:  foreach i ∈{0, ... ,length − 1} do
3:      baydarr[blockID][threadID][i] <- (decimal & 1)
4:      decimal <- (decimal >> 1)
5:  end
}
```

Figure 7 is the performance ratio between INI_BD2 and INI_BD1 for generation of bay divisions. The INI_BD2 uses bitwise operators instead of a lot of mod operators of INI_BD1, yet GPU is not good at such mod calculations due to hardware limitations. So the performance ratio of INI_BD2 is much better than INI_BD1, and with the increase in the number of facilities, the more obvious the performance improvement.

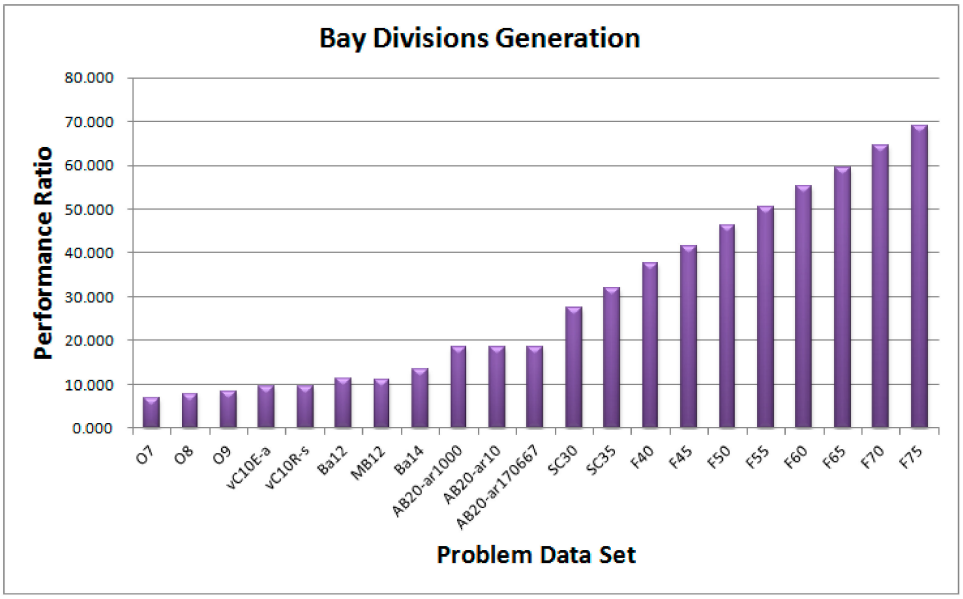

Figure 7. Performance ratio between INI_BD2 and INI_BD1 for generation of bay divisions.

3.3. Parallel Selection on GPU

Selection of GA can be performed in many ways, of which the tournament selection and roulette wheel selection are most famous. Limmer and Fey suggested that selection and recombination operators should be grouped together to investigate their relationship [22]. They adopted tournament selection because one-point crossover for the recombination was used in their work [22]. The one-point crossover produces two offspring from two parent individuals. In the UA-FLP, facility sequences and bay divisions are both required to perform selection and recombination, respectively. According to [21], two different crossover methods are applied: partially mapped crossover (PMX) is used to recombine facility sequences, and n-point crossover method is for recombination of bay divisions. The above feature is quite different from that requiring only one crossover strategy. Therefore, we developed two versions of selection strategies—the tournament selection, and the roulette wheel selection—to investigate which is better for UA-FLP.

The tournament selection involves running several 'tournaments' among a few individuals who are chosen at random from the population, and the winner of each tournament is selected. In our program, each time several pairs of individuals are randomly chosen from the population for a competition. After a series of selections, the individual with the best fitness from these pairs is selected. For parallel tournament selection, our method (SEL_TN) let one thread execute a series of selections to determine one best individual. The time complexity of parallel tournament selection is equal to O($nlogn$).

Roulette wheel selection [22,27], also known as fitness proportionate selection, is for selecting potentially useful solutions by firstly calculating the fitness level for each chromosome and then using fitness levels to associate each individual chromosome with a probability for selection. If f_i is the fitness value of individual i and n is the number of individuals in the population, the probability of the individual being selected is $prob_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$. The probabilities of all individuals are accumulated one by one, where the initial and resultant accumulated values are 0 and 1, respectively. During the accumulation, each individual is associated with two numbers: starting and ending numbers. An individual is selected only when a random number, ranging from 0 to 1, is generated and its value is between the associated starting and ending numbers.

The way for parallelizing roulette wheel selection (SEL_RL) is that one thread is responsible for picking up one better individual in an island. Furthermore, in order to take full advantage of threads

in every block, as shown in Figure 8, important data are stored in shared memory to increase access speed. Shared memory, accessible by the threads in a same block, is high-speed memory. Kernel Algorithm 3 shows the pseudocode to the SEL_RL, i.e., the roulette wheel selection. In the algorithm, shared memory is used to store the total cost of the island, probabilities for individuals, and the sum proportion for each island. Threads in one block carry out selections of individuals for next generation, respectively. With the fast-shared memory to communicate with each other, the system performance is improved. The time complexity of parallel roulette wheel selection is equal to O(n).

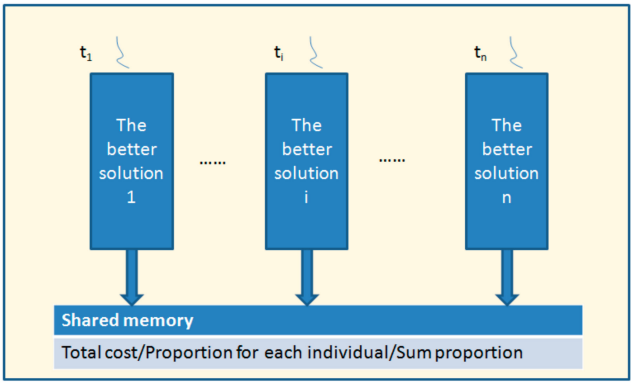

Figure 8. Parallel roulette wheel selection of SEL_RL.

Algorithm 3: Parallel Roulette Wheel Selection

```
Data:
    carr:    Array recording the cost of each facsarr and baydarr pair
    survive: Array recording the picked facsarr and baydarr pair id
    fact:    Variable for total cost of the island
    pmatrix: Array recording proportion that each facsarr and baydarr holds
    total:   Variable for the sum of each island's pmatrix
    base:    Variable for the sum number to calculate elements in pmatrix
Input:
    carr; survive;
Result:
    Individuals picked from current population
Kernel function:
    __global__ void wheel-selection (Array carr, Array survive){
      1:  total <- 0
      2:  fact <- 0
      3:  if threadID == 0 do
      4:    foreach i ∈{0, . . . , pop − 1} do
      5:      fact <- fact + carr[blockID][i]
      6:    end
      7:  end
      8:  pmatrix[blockID][threadID] <- (1/carr[blockID][threadID]) * fact
      9:  if threadID == 0 do
     10:    foreach i ∈{0, . . . , pop − 1} do
     11:      total <- total + pmatrix[blockID][i]
     12:    end
     13:  end
     14:  pick <- Random() mod total
     15:  base <- 0
     16:  foreach i ∈{0, . . . , pop − 1} do
     17:    base <- base + pmatrix[i]
     18:    if base > pick do
     19:      survive[blockID][threadID] = i
     20:    end
     21:  end
    }
```

Figure 9 illustrates the performance ratio between the roulette wheel selection SEL_RL and the tournament selection SEL_TN. The performance of SEL_RL has great performance improvement when it combines with the PMX crossover. In other words, roulette wheel selection is more suitable for UA-FLP due to the crossover strategies adopted, which is different from that in the reference [22].

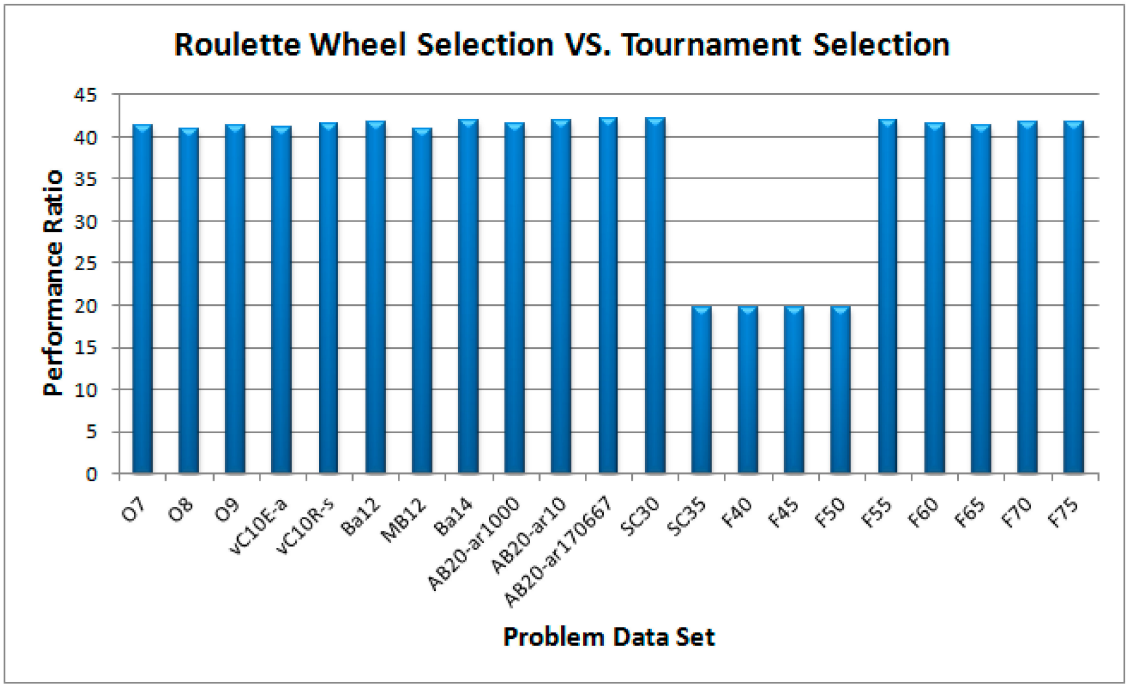

Figure 9. Performance ratio between SEL_RL and SEL_TN for selection operator.

3.4. Parallel Mutation on GPU

Gaussian mutation is applied and it consists mainly of the generation of random numbers in the work [22]. In our GPU solution, Gaussian mutation is also used in both mutations of facility sequence and bay divisions according to the mutation probability. To parallelize the mutation operator, according to the previous work, each thread will generate a random number to determine whether the mutation operator is performed. For the individuals who are mutated, the corresponding threads need to do the mutation operator for their own facility sequence and flexible bay divisions in parallel. Two positions are generated randomly to swap each other for a facility sequence in mutation. In this process, two identical positions are likely to be produced, causing a failed mutation. To avoid this problem, two methods can be adopted. One method (MUT_M1) is regeneration when inspecting the repetition, and the other (MUT_M2) is producing a position for mutation randomly and exchanging the position with its adjacent location. Though the time complexities of MUT_M2 and MUT_M1 are both constant time, MUT_M2 need not check and avoid duplication. Kernel Algorithm 4 shows the pseudocode of the mutation operator for MUT_M2.

Algorithm 4: Parallel Mutation Operator

```
Data:
   bound: Variable for determining if the thread should execute mutation
   mrate: Constant for mutation probability
Input:
   facsarr; baydarr
Result:
   Individuals after mutation
Kernel function:
__global__ void mutation(Array facsarr, Array baydarr) {
1:   bound <- Random() mod 100
2:   if bound <= mrate * 100 do
3:   pos <- Random() mod (length - 1)
4:   Swap(facsarr[blockID][threadID][pos + 1], facsarr[blockID][threadID][pos])
5:   Swap (baydarr[blockID][threadID][pos + 1], baydarr[blockID][threadID][pos])
6:      end
}
```

Figure 10 shows the performance ratio between MUT_M2 and MUT_M1 for mutation. The performance of MUT_M2 is better than MUT_M1 to most of the benchmarks. However, the performance difference is not significant, although theoretically MUT_M2 is more efficient than MUT_M1 because MUT_M2 needs not to check and avoid duplication. The reason is that the mutation probability is not high and it has a small chance to generate two identical locations even when a mutation is required to be performed.

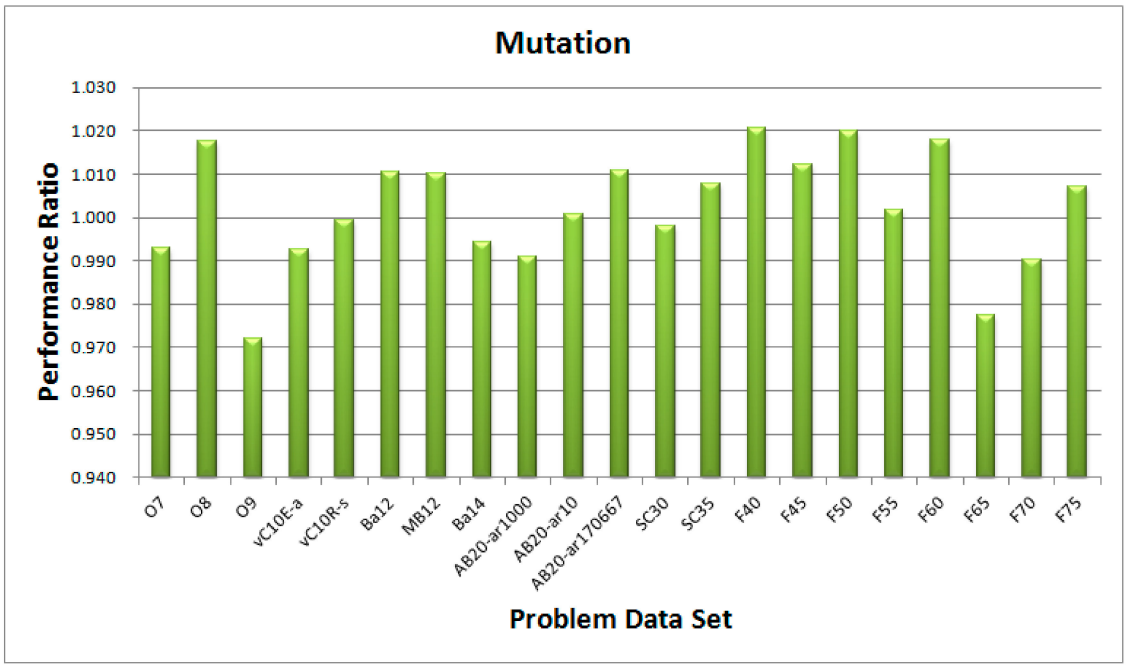

Figure 10. Performance ratio between MUT_M2 and MUT_M1 for mutation.

3.5. Parallel Migration on GPU

The migration operator, the featuring operator in IMGA, exchanges individuals among islands, which provides greater population diversity. With a ring topology, individuals from an island can migrate to their neighboring island during one migration interval. The previous work [22] proposed

that one thread is responsible for migrating an individual (MIG_I). Figure 11 illustrates how their main concept is implemented on GPU, assuming every island needs to migrate two individuals. Each island will be processed by one thread block. Firstly, two individuals, represented with two chromosomes, are selected randomly for each island. Secondly, an island will use two threads to read two chromosomes in its neighboring island from global memory and then write the two chromosomes into its shared memory. Thirdly, synchronization between thread blocks is needed to ensure that the second step is completed by all blocks before any thread block proceeds to next step. Finally, each island uses the two migrated chromosomes in its shared memory to replace its two original chromosomes in global memory, which are executed by two threads with copy operations.

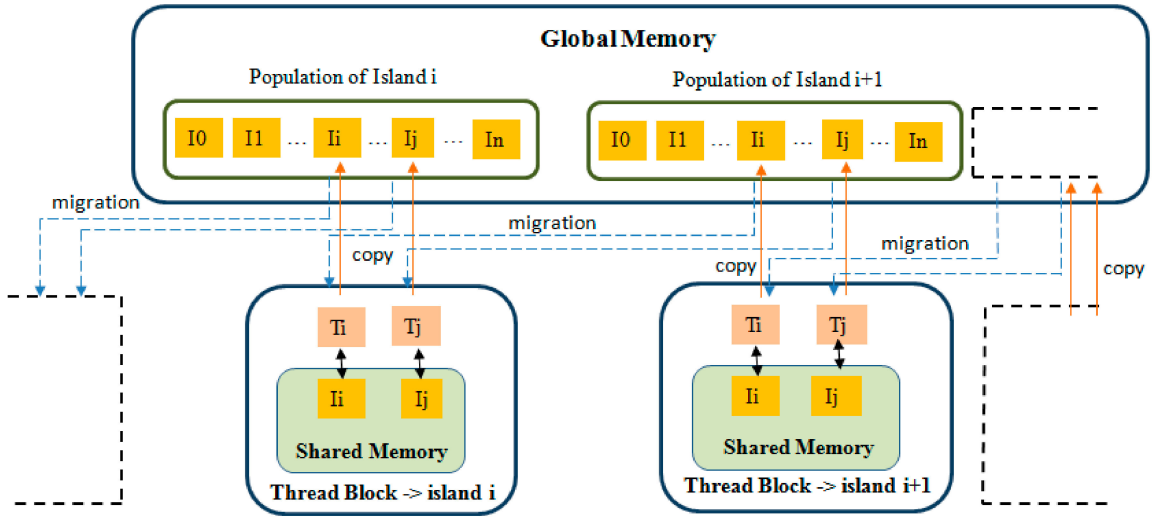

Figure 11. Migration between islands with one thread processing one chromosome (MIG_I).

However, because one chromosome consists of many genes, to increase the parallelism, we propose an enhanced method that uses one thread to deal with one gene, instead of one chromosome, to improve the migrated performance (MIG_G). In the case of UA-FLP, p ($p \geq 7$) facilities—encoded with a gene for each facility—are combined into an individual that represents one facility sequence. Assume that two random individuals are selected for migration, there will be $2p$ threads responsible for the migration. If the number of facilities is large and the migration rate is high, then our method can use more threads to process migration concurrently, which can save more time than the previous method. Figure 12 shows the procedures of how to apply one thread to process one gene in the migration step. The pseudocode of MIG_G is shown in Kernel Algorithm 5.

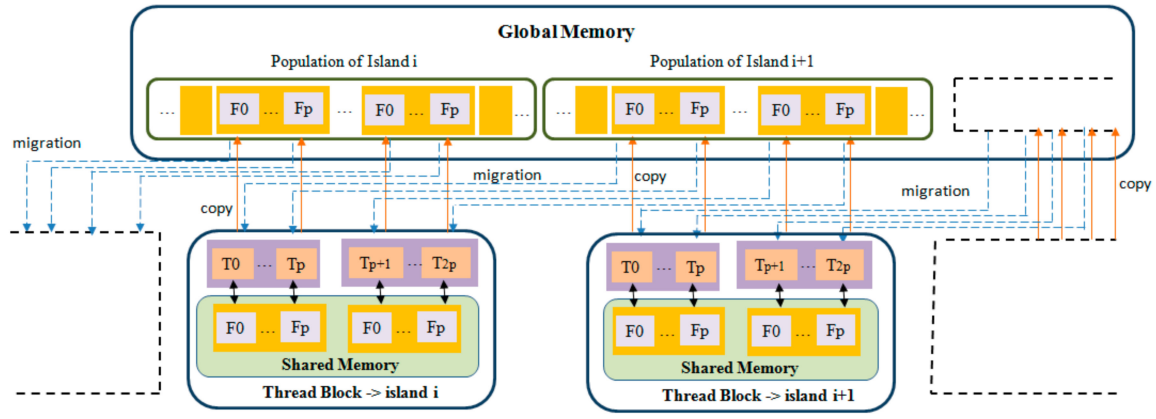

Figure 12. Migration between islands with one thread processing one gene (MIG_G).

Algorithm 5: Parallel Migration Operator

```
Data:
  blockID: Block id
  threadID.x: Thread id
  threadID.y: Thread id
  migiidx: Index of target island for individual migration
  midxarr: Integer array containing index number for each migrated individual
  facsarr: Integer array containing facility sequence for each individual
  baydarr: Boolean array containing flexible bay divisions for each individual
  facsmarr: Integer array containing facility sequence for each migrated individual
  baydmarr: Boolean array containing flexible bay divisions for each migrated individual
Input:
  midxarr, facsarr, baydarr, facsmarr, baydmarr
Result:
  Individuals after mutation
Kernel function:
__global__ void migration (Array midxarr, Array facsarr, Array baydarr, Array facsmarr, Array baymarr){
1.    if (threadID.x == 0) do
2.        midxarr[blockID][threadID.y] <- Random() mod (LENGTH_OF_ISLAND)
3.    end
4.    __syncthreads()
5.    migiidx = (blockID + 1) mod NUMBER_OF_ISLAND
6.    facsmarr[blockID][threadID.y][threadID.x] <-
                    facsarr[migiidx][ midxarr[migiidx][threadID.y] ][threadID.x]
7.    baydmarr[blockID][threadID.y][threadID.x] <-
                    baydarr[migiidx][ midxarr[migiidx][threadID.y] ][threadID.x]
8.    __syncblocks()
9.    facsarr[blockID][ midxarr[blockID][threadID.y] ][threadID.x] <-
                    facsmarr[blockID][threadID.y][threadID.x]
10.   baydarr[blockID][ midxarr[blockID][threadID.y] ][threadID.x] <-
                    baydmarr[blockID][threadID.y][threadID.x]
}
```

According to the experimental result between our proposed method MIG_G and the previous method MIG_I, as shown in Figure 13, the performance ratio of MIG_G is up to more than 1.6 for exploring more parallelisms. If the migration rate is higher, the efficiency of MIG_G is also better.

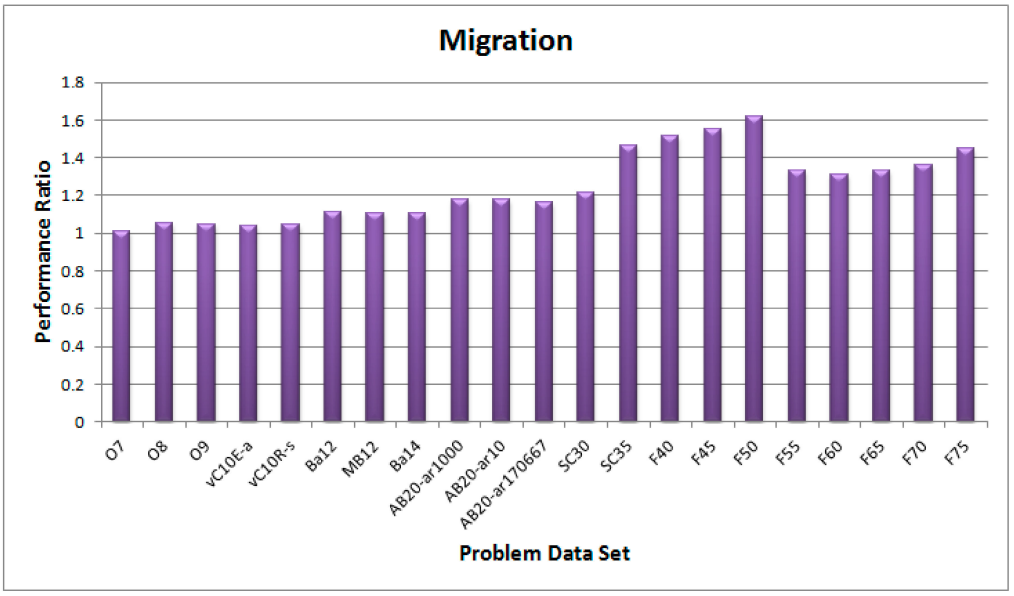

Figure 13. Performance ratio between MIG_G and MIG_I for migration.

The reason why using each thread to process a gene is better than using the thread to process a chromosome is because we not only take full advantage of the fine-grained parallelism, but also utilize memory coalescing to access global memory in our improved migration algorithm. Accessing data in global memory is critical to the performance of a CUDA application since read and write operations are slow in global memory that is comprised of dynamic random-access memories (DRAMs). A parallel process of modern DRAMs is that for each time when a location is accessed, many consecutive locations including the requested location are accessed. Therefore, only in this way of parallel accessing can DRAMs work close to the advertised peak global memory bandwidth if an application uses data from consecutive locations. Figure 14 shows the memory coalescing of our MIG_G. When all threads in a warp execute a same instruction to read the migrated genes, the hardware detects whether the threads access consecutive memory locations in global memory. Because the genes of a chromosome are stored in an array with consecutive locations in global memory, the hardware coalesces all memory accesses into one consolidated access to these consecutive DRAM locations. While in the preliminary algorithm MIG_I, one thread migrates one chromosome. Although each chromosome is adjacent to each other, a stride exists between them as there are many genes in each chromosome. In a CUDA GPU, the threads in a same warp will access genes at the same place in their responsible chromosomes. For instance, if one thread access to the first gene, the other threads in the same warp will also access to their first gene in their responsible chromosomes simultaneously. However, the first genes are not consecutive. When all the threads in a warp execute a read operation to the chromosomes in global memory, usually more than one memory access transaction is required, resulting in a performance worse than that with memory coalescing, like our proposed algorithm. The complexity of MIG_I is O(p) as each thread must move the genes sequentially, while the complexity of MIG_G is O(1) because one thread moves one gene.

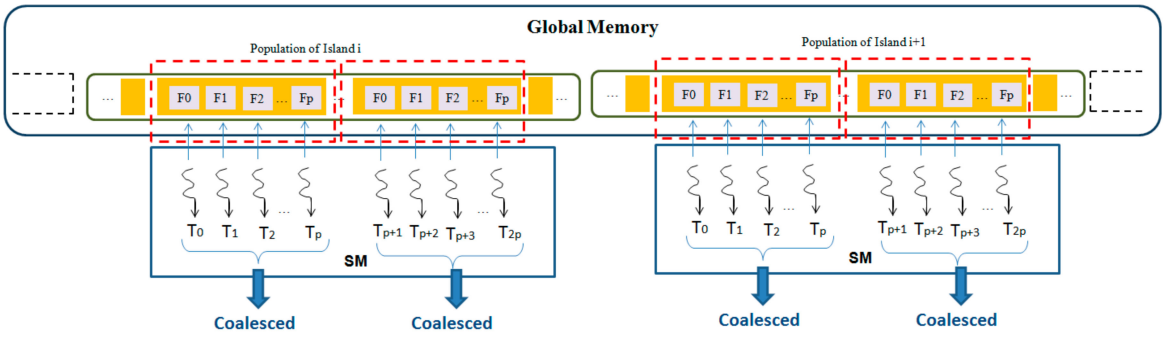

Figure 14. Memory coalescing represents a data chunk.

To investigate how the number of threads per block and the number of blocks influence the performance of our migration algorithm, we have conducted an experiment and the results are shown in Figure 15. In our method, each island is processed by one thread block and one gene is moved by one thread. In this experiment, we assume five chromosomes are migrated per island. Consequently, the number of blocks is equal to the number of islands, and the number of threads per block is equal to the product of the number of chromosomes and the number of genes per chromosome. Moreover, because the number of genes per chromosome is equal to the number of facilities, we choose the benchmarks based on the numbers of genes per chromosome required in the experiment. The migration workloads are kept the same for all cases. The execution times are normalized to the execution time of the case with 15 blocks and 300 threads. When the number of threads is increased, the execution time is decreased except the case with 90 blocks. If there are 90 blocks, the number of threads per block is down to 50, indicating that only two warps of threads can be used to exploit warp-level parallelism. Nevertheless, the execution time differences between any two of them are insignificant.

3.6. Parallel UA-FLP Integrated with Better Strategies

According to the comparison of two alternative parallelization strategies for each main step, the better strategies are chosen and integrated into the complete algorithm of IMGA for solving UA-FLP on GPU, which is our improved GPU version. For comparison, another GPU version is also implemented: called the previous version, which consists of conventional and previously proposed parallelization methods. Figure 16 shows the execution times of our GPU version, the previous GPU version, and the CPU version for the larger data sets. GPU versions outperform the CPU version significantly all the time. Our GPU version always requires the shortest execution time for any data set.

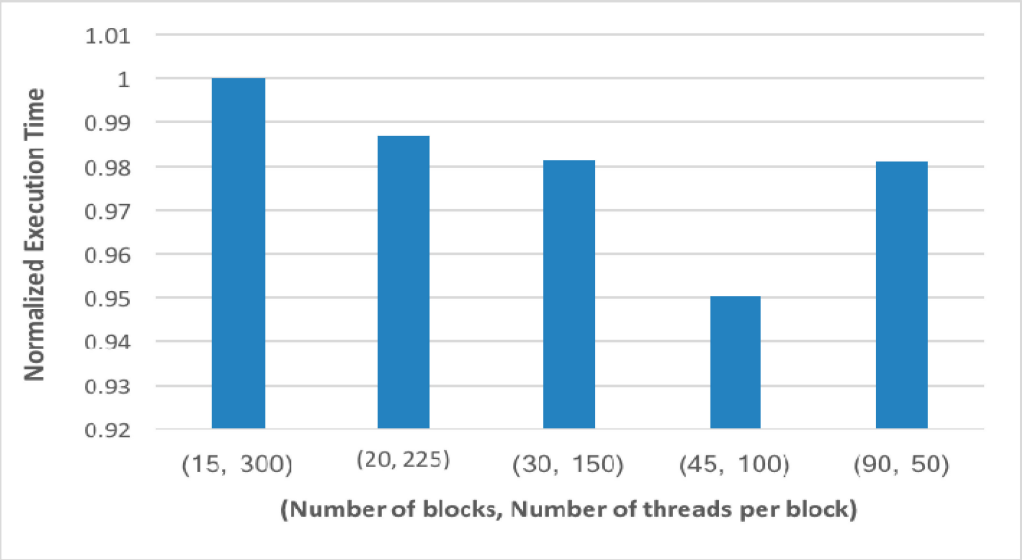

Figure 15. Comparison of normalized execution time for various combinations of number of blocks and number of threads per block. The number of blocks is equal to the number of islands. The number of threads per block is equal to the product of the number of chromosomes and the number of genes per chromosome. In this comparison, five chromosomes are migrated per island.

We compare the performance improvements executed on the CPU and GPU for all 34 data sets, as shown in Figure 17, which indicates the performance ratios of our improved GPU version over the CPU version and the performance ratios of the previous GPU version over the CPU version. Note that the data sets are listed in ascending order based on the number of facilities. The previous GPU version outperforms the CPU one for all the data sets. The best performance ratio is up to 20. On the other hand, our GPU version provides much better performance for all cases, with the performance ratio over the CPU version up to 84. Comparing the performance ratios provided by two versions of GPU, our GPU version is much better, meaning the better parallelization methods we suggest for IMGA steps really contribute to performance improvement. Moreover, when the number of facilities is increased, the performance of our GPU version is also highly improved.

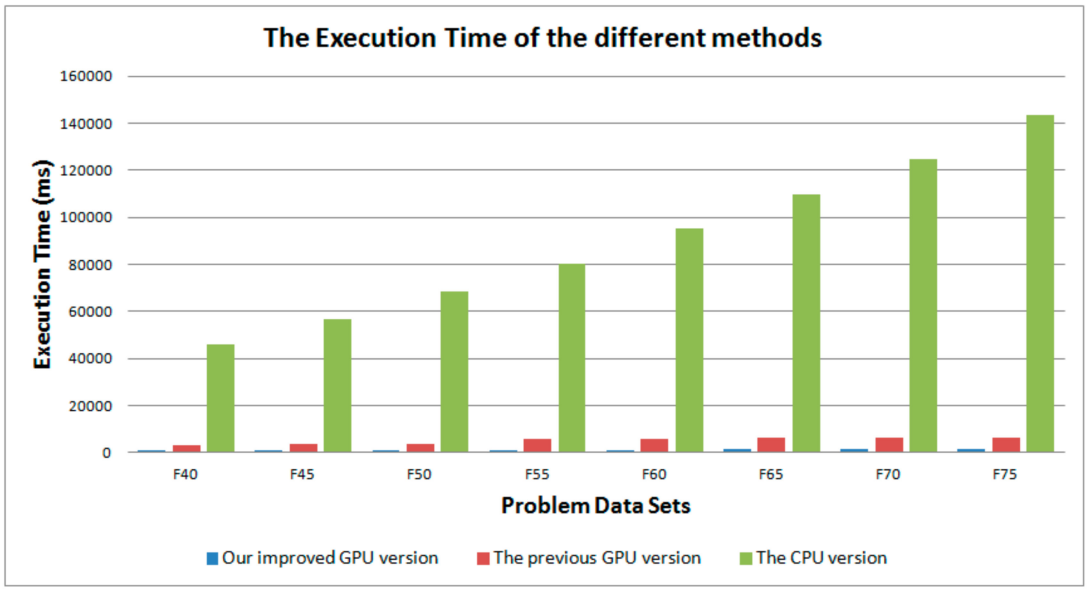

Figure 16. Execution times of the different methods.

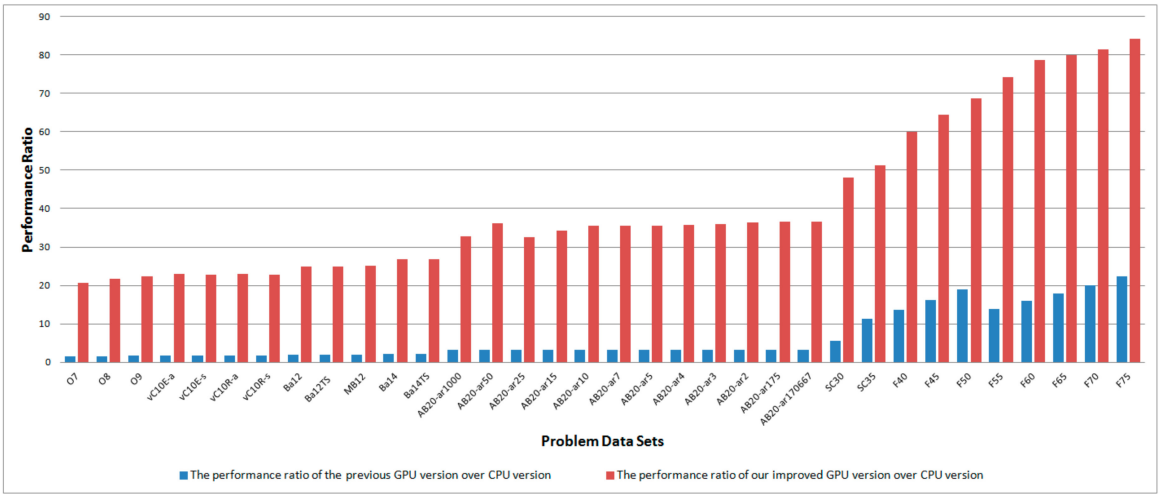

Figure 17. Comparisons of performance ratio between two GPU versions and CPU version, respectively. The previous GPU version is implemented with previous parallel methods while our GPU version is implemented with our proposed methods.

4. Conclusions

UA-FLPs are widely used in different fields, including industrial facility design, warehouse organization, school facility layout, and VLSI element placement. However, UA-FLP is an NP-hard problem. With the increase in the problem size, the amount of calculation will become larger and larger, and the execution time is getting longer and longer even though metaheuristic approaches such as GA are applied to solve UA-FLP. In the traditional CPU workstations, when the number of the facilities reaches a certain level, it is difficult to calculate the appropriate solution within a reasonable period of time. IMGA, one of the most promising variants among the parallelized models of GA, can explore different regions of the search space. IMGA can improve the solution for most of the problem data set for the UA-FLP [21]. However, the IMGA method for UA-FLP still requires long execution time, so that the authors pointed an interesting future work: parallelizing IMGA.

Using GPU to solve large-scale and complex computing has been widely investigated. Modern GPU, as specialized computer processors, can manipulate large amounts of data efficiently by

highly parallel many-core system. Parallel IMGA on GPU is focused in the recent work [22], while their parallelization approaches are preliminary because the authors mainly emphasize comparing performances between different parallel architectures. In addition, they used only one mathematical function to model and implement the IMGA.

In this paper, we further investigated on parallelizing IMGA by GPU. We proposed multiple parallel algorithms for the main steps in the IMGA when solving the UA-FLP, including initialization, selection, mutation, and migration operators. In data initialization, there are two parts: generations of facility sequence and bay divisions. To generate the facility sequence, we proposed an improved method to eliminate a large number of comparisons in conventional method. Our method can also avoid regeneration of random numbers for the same array element. For the above reasons, our approach can reduce the time complexity from the previous $O(p^2)$ to $O(p)$. To generate bay divisions, our improved method used bitwise operators, instead of a lot of module operators in conventional methods, resulting in the time complexity is greatly reduced from $O(p)$ to $O(1)$. According to the previous work [22], when choosing a selection operator, we should take into account the relationship between it and the recombination operator. Therefore, we implemented two selection strategies—tournament selection and roulette wheel selection—to investigate which is better for UA-FLP. Through experimental results, roulette wheel selection is more suitable for UA-FLP due to the crossover strategies adopted, which is quite different from that in the reference [22] because two different crossover strategies are used for FBS encoding in UA-FLP. To parallelize the mutation operator, our method needs not to check for duplications, which improved the mutation performance. For migration parallelization, we proposed an enhanced method that uses one thread to deal with one gene, instead of one chromosome, to improve the migrated performance. Our improved migration algorithm is better than the conventional one because we take full advantage of the fine-grained parallelism and utilize memory coalescing to access global memory.

Experiments were conducted to compare performances between the different parallel algorithms for each step. Based on theoretical derivation and experimental results, we integrated the better parallel method in each step into the parallel IMGA for UA-FLP on GPU. The experiments were conducted on a total of 26 well-known UA-FLPs data sets and 8 additional benchmarks for large-scale facilities. According to the experimental results, if conventional parallelization methods for main steps of IMGA are applied to GPU implementation, the performance ratio over the CPU version is more than 20 at best. However, if our suggested parallel methods are used to implement IMGA on GPU, the best performance ratio over the CPU version can be as high as 84. In other words, our GPU version provides much higher performance than conventional GPU version.

In our future work, more effective methods can be proposed to optimize parallel IMGA, such as how to manage memories, improve fitness evaluations, etc.

Author Contributions: Conceptualization, X.S., L.-F.L., P.C. and C.-C.W.; Data curation, X.S. and P.C.; Formal analysis, X.S., P.C., and C.-C.W.; Funding acquisition, X.S., L.-F.L., and C.-C.W.; Investigation, X.S., L.-F.L., P.C., and C.-C.W.; Methodology, X.S., P.C., and C.-C.W.; Project administration, L.-F.L. and C.-C.W.; Resources, X.S.; Software, X.S., P.C., and C.-C.W.; Supervision, C.-C.W.; Validation, X.S. and C.-C.W.; Writing—original draft, X.S., L.-F.L., L.-R.C., and C.-C.W.; Writing—review & editing, X.S., L.-F.L., L.-R.C., and C.-C.W.

Funding: This research is supported by Ministry of Science and Technology, Taiwan (grant No. MOST 106-2221-E-018-010, MOST 106-2221-E-018-009), and Beijing Municipal Commission of Education (grant No. KM201811417010).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Drira, A.; Pierreval, H.; Hajri-Gabouj, S. Facility layout problems: A survey. *Annu. Rev. Control* **2007**, *31*, 255–267. [CrossRef]
2. Tompkins, J.A.; White, J.A.; Bozer, Y.A.; Tanchoco, J.M.A. *Facilities Planning*; John Wiley & Sons: New York, NY, USA, 2010.

3. Meller, R.D.; Gau, K.Y. The facility layout problem: Recent and emerging trends and perspectives. *J. Manuf. Syst.* **1996**, *15*, 351–366. [CrossRef]
4. Scholz, D.; Petrick, A.; Domschke, W. STaTS: A slicing tree and tabu search based heuristic for the unequal area facility layout problem. *Eur. J. Oper. Res.* **2009**, *197*, 166–178. [CrossRef]
5. Kusiak, A.; Heragu, S.S. The facility layout problem. *Eur. J. Oper. Res.* **1987**, *29*, 229–251. [CrossRef]
6. Garey, M.R. *Computers and Intractability: A Guide to the Theory of NP-Completeness*; WH Free Co.: San Francisco, CA, USA, 1979; pp. 90–91.
7. Wang, M.J.; Hu, M.H.; Ku, M.Y. A solution to the unequal area facilities layout problem by genetic algorithm. *Comput. Ind.* **2005**, *56*, 207–220. [CrossRef]
8. Wong, K.Y. Solving facility layout problems using Flexible Bay Structure representation and Ant System algorithm. *Expert Syst. Appl.* **2010**, *37*, 5523–5527. [CrossRef]
9. Kulturel-Konak, S.; Konak, A. Unequal area flexible bay facility layout using ant colony optimisation. *Int. J. Prod. Res.* **2010**, *49*, 1877–1902. [CrossRef]
10. Xiao, Y.J.; Zheng, Y.; Zhang, L.M.; Kuo, Y.H. A combined zone-LP and simulated annealing algorithm for unequal-area facility layout problem. *Adv. Prod. Eng. Manag.* **2016**, *11*, 259. [CrossRef]
11. Aiello, G.; La Scalia, G.; Enea, M. A non dominated ranking Multi Objective Genetic Algorithm and electre method for unequal area facility layout problems. *Expert Syst. Appl.* **2013**, *40*, 4812–4819. [CrossRef]
12. Aiello, G.; Scalia, G.L.; Enea, M. A multi objective genetic algorithm for the facility layout problem based upon slicing structure encoding. *Expert Syst. Appl.* **2012**, *39*, 10352–10358. [CrossRef]
13. Holland, J.H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Application to Biology, Control, and Artificial Intelligence*; University of Michigan Press: Ann Arbor, MI, USA, 1975.
14. Kenneth, A.D.J. An Analysis of the Behavior of a Class of Genetic Adaptive Systems. Ph.D. Dissertation, University of Michigan, Ann Arbor, MI, USA, 1975.
15. Darwin, C. *The Origin of Species by Means of Natural Selection, or, the Preservation of Favoured Races in the Struggle for Life*; With a Foreward by George Gaylord Simpson; Collier Books: New York, NY, USA, 1962.
16. Tan-Hsu, T.; Bor-An, C.; Yung-Fa, H. Performance of Resource Allocation in Device-to-Device Communication Systems Based on Evolutionally Optimization Algorithms. *Appl. Sci.* **2018**, *8*, 1271. [CrossRef]
17. Perez-Ramirez, C.A.; Jaen-Cuellar, A.Y.; Valtierra-Rodriguez, M.; Dominguez-Gonzalez, A.; Osornio-Rios, R.A.; Romero-Troncoso, R.D.J.; Amezquita-Sanchez, J.P. A two-step strategy for system identification of civil structures for Structural Health Monitoring using wavelet transform and genetic algorithms. *Appl. Sci.* **2017**, *7*, 111. [CrossRef]
18. Kuo, C.C.; Liu, C.H.; Chang, H.C.; Lin, K.J. Implementation of a motor diagnosis system for rotor failure using genetic algorithm and fuzzy classification. *Appl. Sci.* **2016**, *7*, 31. [CrossRef]
19. Montazeri, A.; West, C.; Monk, S.D.; Taylor, C.J. Dynamic modelling and parameter estimation of a hydraulic robot manipulator using a multi-objective genetic algorithm. *Int. J. Control* **2017**, *90*, 661–683. [CrossRef]
20. Shin, H.; Joo, C.; Koo, J. Optimal rehabilitation model for water pipeline systems with genetic algorithm. *Procedia Eng.* **2016**, *154*, 384–390. [CrossRef]
21. Palomo-Romero, J.M.; Salas-Morera, L.; García-Hernández, L. An island model genetic algorithm for unequal area facility layout problems. *Expert Syst. Appl.* **2017**, *68*, 151–162. [CrossRef]
22. Limmer, S.; Fey, D. Comparison of common parallel architectures for the execution of the island model and the global parallelization of evolutionary algorithms. *Concurr. Comput. Pract. Exp.* **2016**, *29*, e3797. [CrossRef]
23. Pospichal, P.; Jaros, J.; Schwarz, J. Parallel Genetic Algorithm on the CUDA Architecture. Applications of Evolutionary Computation. In Proceedings of the Evoapplicatons 2010: Evocomplex, Evogames, Evoiasp, Evointelligence, Evonum, and Evostoc, Istanbul, Turkey, 7–9 April 2010; pp. 442–451.
24. Moumen, Y.; Abdoun, O.; Daanoun, A. Parallel approach for genetic algorithm to solve the Asymmetric Traveling Salesman Problems. In Proceedings of the 2nd International Conference on Computing and Wireless Communication Systems, Larache, Morocco, 14–16 November 2017; ACM: New York, NY, USA, 2017.
25. Abdelkafi, O.; Idoumghar, L.; Lepagnot, J.; Paillaud, J.L.; Deroche, I.; Baumes, L.; Collet, P. Using a novel parallel genetic hybrid algorithm to generate and determine new zeolite frameworks. *Comput. Chem. Eng.* **2017**, *98*, 50–60. [CrossRef]

26. Melab, N.; Talbi, E.G. GPU-based island model for evolutionary algorithms. In Proceedings of the 12th annual conference on Genetic and evolutionary computation, Porland, OR, USA, 7–11 July 2010; ACM: New York, NY, USA, 2010.
27. Shojafar, M.; Cordeschi, N.; Baccarelli, E. Energy-efficient adaptive resource management for real-time vehicular cloud services. *IEEE Trans. Cloud Comput.* **2016**, *99*, 1–14. [CrossRef]
28. Shojafar, M.; Canali, C.; Lancellotti, R.; Abawajy, J. Adaptive computing-plus-communication optimization framework for multimedia processing in cloud systems. *IEEE Trans. Cloud Comput.* **2016**, *99*, 1–14. [CrossRef]
29. Javanmardi, S.; Shojafar, M.; Shariatmadari, S.; Abawajy, J.H.; Singhal, M. PGSW-OS: A novel approach for resource management in a semantic web operating system based on a P2P grid architecture. *J. Supercomput.* **2014**, *69*, 955–975. [CrossRef]
30. Li, C.C.; Lin, C.H.; Liu, J.C. Parallel genetic algorithms on the graphics processing units using island model and simulated annealing. *Adv. Mech. Eng.* **2017**, *9*. [CrossRef]
31. Bonelli, F.; Tuttafesta, M.; Colonna, G.; Cutrone, L.; Pascazio, G. An MPI-CUDA approach for hypersonic flows with detailed state-to-state air kinetics using a GPU cluster. *Comput. Phys. Commun.* **2017**, *219*, 178–195. [CrossRef]
32. Chen, J.H.; Chen, R.C.; Liu, J.L. A GPU Poisson–Fermi solver for ion channel simulations. *Comput. Phys. Commun.* **2018**, *229*, 99–105. [CrossRef]
33. Madhikar, P.; Åström, J.; Westerholm, J.; Karttunen, M. CellSim3D: GPU accelerated software for simulations of cellular growth and division in three dimensions. *Comput. Phys. Commun.* **2018**, *232*, 206–213. [CrossRef]
34. Kim, S.; Cho, J.; Park, D. Moving-target position estimation using GPU-based particle filter for iot sensing applications. *Appl. Sci.* **2017**, *7*, 1152. [CrossRef]
35. Han, T.D.; Abdelrahman, T.S. hiCUDA: High-level GPGPU programming. *IEEE Trans. Parallel Distrib. Syst.* **2010**, *22*, 78–90. [CrossRef]
36. Han, T.D.; Abdelrahman, T.S. Reducing branch divergence in GPU programs. In Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, Newport Beach, CA, USA, 5 March 2011; ACM: New York, NY, USA, 2011.
37. Armour, G.C.; Buffa, E.S. A heuristic algorithm and simulation approach to relative location of facilities. *Manag. Sci.* **1963**, *9*, 294–309. [CrossRef]
38. Aiello, G.; Enea, M. Fuzzy approach to the robust facility layout in uncertain production environments. *Int. J. Prod. Res.* **2001**, *39*, 4089–4101. [CrossRef]
39. Tate, D.M.; Smith, A.E. Unequal-area facility layout by genetic search. *IIE Trans.* **1995**, *27*, 465–472. [CrossRef]
40. NVIDIA (2014) Whitepaper NVIDIA GeForce GTX 980. Available online: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF (accessed on 11 August 2018).