symmetry

MDPI

Article

Quality-Oriented Study on Mapping Island Model Genetic Algorithm onto CUDA GPU

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

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 Management Information Systems, National Chengchi University, Taipei 11605, Taiwan; littlesixdot@gmail.com
4 Department of Computer Science and Information Engineering, National Changhua University of Education, Changhua 50007, Taiwan
* Correspondence: ccwu@cc.ncue.edu.tw; Tel.: +886-4-7232105 (ext. 8431)

Received: 26 January 2019; Accepted: 21 February 2019; Published: 2 March 2019

Abstract: Genetic algorithm (GA), a global search method, has widespread applications in various fields. One very promising variant model of GA is the island model GA (IMGA) that introduces the key idea of migration to explore a wider search space. Migration will exchange chromosomes between islands, resulting in better-quality solutions. However, IMGA takes a long time to solve the large-scale NP-hard problems. In order to shorten the computation time, modern graphic process unit (GPU), as highly-parallel architecture, has been widely adopted in order to accelerate the execution of NP-hard algorithms. However, most previous studies on GPUs are focused on performance only, because the found solution qualities of the CPU and the GPU implementation of the same method are exactly the same. Therefore, it is usually previous work that did not report on quality. In this paper, we investigate how to find a better solution within a reasonable time when parallelizing IMGA on GPU, and we take the UA-FLP as a study example. Firstly, we propose an efficient approach of parallel tournament selection operator on GPU to achieve a better solution quality in a shorter amount of time. Secondly, we focus on how to tune three important parameters of IMGA to obtain a better solution efficiently, including the number of islands, the number of generations, and the number of chromosomes. In particular, different parameters have a different impact on solution quality improvement and execution time increment. We address the challenge of how to trade off between solution quality and execution time for these parameters. Finally, experiments and statistics are conducted to help researchers set parameters more efficiently to obtain better solutions when GPUs are used to accelerate IMGA. It has been observed that the order of influence on solution quality is: The number of chromosomes, the number of generations, and the number of islands, which can guide users to obtain better solutions efficiently with moderate increment of execution time. Furthermore, if we give higher priority on reducing execution time on GPU, the quality of the best solution can be improved by about 3%, with an acceleration that is 29 times faster than the CPU counterpart, after applying our suggested parameter settings. However, if we give solution quality a higher priority, i.e., the GPU execution time is close to the CPU's, the solution quality can be improved up to 8%.

Keywords: genetic algorithm; island model; unequal area facility layout problem; quality

1. Introduction

Many metaheuristic algorithms have been proposed to obtain near-optimal solution in finite time when solving NP-hard problems [1–5]. Among them is the genetic algorithm (GA), which is one of the most popular and widely studied algorithms, and is originally proposed by Holland [6] and DeJong [7]. GA is a powerful, domain-independent search technique inspired by Darwinian Theory [8]. As a global search algorithm, in many years GA has been widely used to help solve many optimization problems in various fields [9–18].

Several GA variants have also been proposed. One of the most promising variant is the island model GA (IMGA) [19]. IMGA is a multiple-population coarse-grained model, consisting of several islands distributed subpopulations for occasional exchanges of individuals. Because of the migration processes of individuals between the independent islands, this model is more likely to explore different search regions for better-quality solutions [20,21]. For instance, Juan M. Palomo-Romero et al. [20] applied IMGA to solve the unequal area facility layout problem (UA-FLP). According to the experimental results, most of the instances have obtained better solutions in fewer generations when using IMGA.

UA-FLP is an important and widely studied facility layout problem in industry engineering [22]. A good placement of facilities can reduce up to 50% of total expenses in manufacturing [23]. The problem is also widely used in many fields such as industrial facility design, warehouse organization, and VLSI placement. Most of the researchers build optimization models with quantitative performance criteria, such as optimization of material handling cost. UA-FLP was proven to be an NP-hard combinatorial optimization problem, meaning that when the number of facilities increases, the calculation time is exponentially increased [24]. More and more metaheuristic algorithms [25–30] are proposed to solve UA-FLP for finding better near-optimal solutions. Among them, GA is the most popular algorithm due to its effectiveness in enhancing the opportunity to achieve global optimal solutions. Lately, the IMGA is proving to be an efficient method to solve UA-FLP, which can obtain good solutions in a reasonable computational time [20].

Although IMGA is effective in solving many NP-hard problems, it takes a much longer execution time on CPU when the problem size becomes larger, since the individuals in IMGA should be selected, evaluated, crossed over, mutated and migrated,. Modern GPUs (Graphics Processing Units) are very popular and flexible processors in high performance computing, especially when CUDA (compute unified device architecture) were distributed [31]. The IMGA can take full advantage of the computing abilities by exploiting coarse and fine grained parallelisms of GPU. However, only a very limited number of studies have focused on how to reduce the execution time of IMGA on GPU [32–34]. In our previous work [32], we have proposed two different parallel algorithms for each individual step of IMGA when solving UA-FLP on GPU. Which parallel algorithm has a better performance for each step has also been reported. The best performance ratio of our suggested GPU version over the CPU counterpart can be as high as 84.

Most previous studies about GPUs are focused on performance, because the searched solutions for the CPU and the GPU implementation of the same method are exactly the same. Therefore, the related work did not usually report the quality. In practice, when implementing IMGA on GPU, settings of various parameters have a great influence on both solution quality and execution time. Since there are many combinations of parameter settings, it is very time consuming to have the best parameter setting. The easiest way to have a better solution is to let key parameters have large values, e.g., the number of generations. However, a larger parameter value usually implies a longer execution time, but it is not for sure that the solution quality will be obviously improved after a parameter is enlarged. To address the problem in this paper, we investigate how each of the key parameters influence the solution quality and the execution time when running IMGA on a GPU. The results of the paper will give us a guideline, by giving a selection order of parameters when users aim at having better solutions, without significantly increasing the execution time.

In this paper, we explore how to obtain a better solution within a reasonable time when parallelizing IMGA on GPU, taking UA-FLP as an example. Firstly, we focus on how to implement the tournament selection in IMGA on a GPU. The key issue is how random numbers are generated by each thread at the runtime on a GPU. Since there are more than thousands of threads that are executed in parallel, the aggregated distribution of all random numbers will influence the actual exploited search space. Next, we investigate three important parameters closely related to the solution quality and the execution time in IMGA, including the number of islands, the number of generations (iterations), and the number of chromosomes. By tuning these parameters separately, the solution quality and the execution time are both influenced. We address the challenge of how to trade off between solution quality and execution time for the parameters. In other words, which parameter is the best to improve the solution quality if the execution time is increased with the same ratio. Experiments are conducted on our selected data sets. Through analysis, the order of the influence of the three parameters on solution quality is observed, which can help researchers adjust parameters in IMGA for finding better qualities on a GPU. Finally we recommend a group of parameters for solving UA-FLP with IMGA on a GPU. According to the experimental results, if we give higher priority on reducing execution time on a GPU, the quality of the best solution can be improved by about 3%, with an acceleration that is 29 times faster than the CPU counterpart after applying our suggested parameter settings. If we give solution quality a higher priority, when the GPU performs almost the same execution time as the CPU, the solution quality can be improved by up to 8%. This research is very valuable for helping users set parameters more efficiently when using a GPU to solve UA-FLP with IMGA, in order to get better solutions in an execution time comparable with that in a CPU. It also helps researchers take advantage of GPUs to find a more appropriate solution for optimization problems.

The paper is organized as follows. Section 2 introduces the related work, including the introduction to UA-FLP, CUDA GPUs, and parallelization of IMGA for UA-FLP on GPU. In Section 3, our improved parallel tournament selection on a GPU is described. Experiments and evaluations about our proposed method and the quality-oriented discussions are illustrated in detail in Section 4. Finally, conclusions are given in Section 5.

2. Related Work

In this section, the related work is introduced, including the formulation and layout representation of UA-FLP, the brief introduction to CUDA GPUs, and GPU-based IMGA for solving UA-FLP.

2.1. UA-FLP

UA-FLP is formulated based on some assumptions. If there is a fixed rectangular region with an area of $W(width) \times H(height)$, where facilities or departments should be located, the objective of the optimization is to minimize the total material handling cost (MFC), which is commonly represented by the sum of the products of the weighted rectilinear distance, and the material handling flow between the centroids over all facility pairs. Assuming the number of facilities is p, the cost of moving materials from facility i to facility j is f_{ij}, and the weighted rectilinear distance between facilities i and j is d_{ij}, the general model is formulated by the following Equation (1).

$$\min MFC = \sum_{i=1}^{p} \sum_{j=1, i \neq j}^{p} f_{ij} d_{ij} \tag{1}$$

The distance d_{ij} can be calculated as Euclidean (Equation (2)) or rectilinear (Equation (3)), where the point defined by x and y is the center of the facility:

$$d_{ij} = \sqrt{(x_j - x_i)^2 + (y_i - y_j)^2} \tag{2}$$

$$d_{ij} = |x_i - x_j| + |y_i - y_j| \tag{3}$$

The general model of UA-FLP is not shape constrained, that is, no minimum side length or maximum aspect ratio is specified for any facility. To ensure the realistic layouts, we imposed a constraint of maximum allowable aspect ratio (α). If the aspect ratio of one facility is more than α, this facility is infeasible. Let W_i and L_i be the length and width of facility i, the aspect ratio of the facility is calculated by Equation (4) below:

$$\alpha = \frac{\max(W_i, L_i)}{\min(W_i, L_i)} \tag{4}$$

The most commonly used continuous representation structure for UA-FLP is the flexible bay structure (FBS), defined by Tong [35]. The layout of the plant is divided into bays of varying widths in one direction. Each bay allows one or more facilities to be located, and its width is variable, depending on the facilities it contains. In our UA-FLP, an individual of facility layout with FBS is divided into two parts that are; (*i*) the facility sequence composed of n organized facilities bay by bay, from left to right and from top to bottom, and (*ii*) bay divisions with $n-1$ binary elements, where value of 0 indicates the facility is placed in the same bay as the previous one, and value of 1 represents the facility is the last facility in the current bay. Figure 1 shows an example of an individual representing with FBS, and its FBS codes are presented in Figure 2.

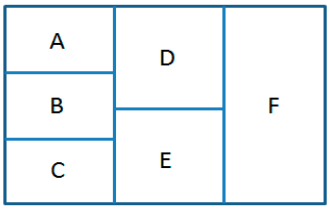

Figure 1. A facility layout example for using FBS representation.

Facility Sequence					
A	B	C	D	E	F

Bay Divisions				
0	0	1	0	1

Figure 2. FBS codes for the example in Figure 1.

2.2. Introduction to GPU and CUDA

With the introduction of the CUDA platform, more and more researchers pay more attention to NVIDIA GPUs since CUDA makes GPU coding easier [36]. A GPU-chip is composed of streaming-multiprocessors (SMs), on each of which there are several streaming processors (SPs). Each SP in the same SM executes the same instruction on different data during each cycle. Figure 3 shows the simple GPU structure:

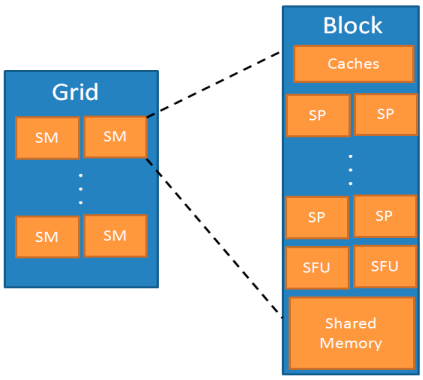

Figure 3. Simple CUDA GPU structure.

A CUDA program consists of sequential segments and parallel segments of codes. The sequential codes run on the host (CPU), and the parallel codes embedded in kernel functions are offloaded to the device (GPU). When a kernel is invoked, a large number of threads are launched to exploit data parallelism after thread blocks are distributed to SMs. After a kernel finishes its execution on the device, the results could be copied from GPU memory to CPU memory. In a CUDA application, threads running the same instructions on different data are grouped into blocks. One active thread runs on an SP, and one block is assigned to an SM. The maximal number of threads in a block is determined by the difference of GPU compute capability. Threads in each block are divided further into Warps. A Warp contains 32 threads in a block, and it adopts the Single Instruction Multiple data (SIMD) execution model, meaning all threads within a warp must execute the same instructions at the instance of any given time. If the threads in a warp are diverged to two paths after a data-dependent conditional branch, the warp serially executes branch paths one by one, resulting in threads becoming idle on another path. It is known as branch divergence, and can degrade the performance of the GPU significantly [37].

There are types of memories residing on a GPU chip, including global memory, shared memory, constant memory, local memory, and register. Different types of memories are differing in size, access scope, access time and whether it is read-only or cached. Figure 4 shows the CUDA memory hierarchy model:

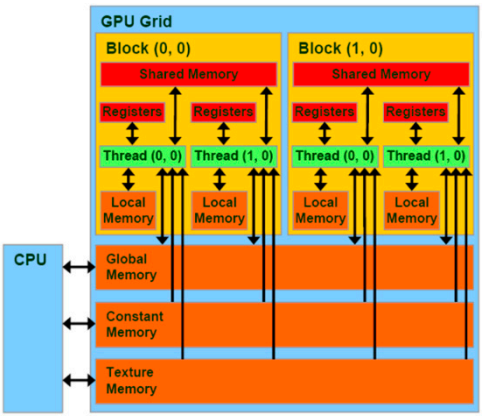

Figure 4. CUDA memories and hierarchy model [38].

Global memory, the largest memory (up to several gigabytes) in a GPU, is used for communication between the host and the device. The access time of global memory is longest, since the latency for completing a read or a write operation for instance, requires 400–800 clock cycles. Memory coalescing is a very important mechanism for hiding such memory latency. The host is responsible for the allocation and deallocation of buffers in the space of the global memory. When a kernel is launched to process, the device takes over the access rights to the buffer in the global memory, until the kernel execution is complete. Through the global memory blocks can communicate with each other.

Shared memory is a very fast memory in a GPU, as it only requires two to four cycles if no bank conflicts exist. It can only be accessed by threads in the same block to communicate with each other. Shared memory is available for both reading and writing, but it is not cached and the memory space is limited.

Constant memory is a read-only and cached memory. A CPU can read and write data in constant memory, but a GPU thread only can read data in it. The constant memory is as fast as the shared memory, unless cache misses occur. Texture memory is similar to constant memory, which is also cached and read-only, but it has a larger memory space and its access latency depends on whether cache misses occur.

Local memory is used to save large automatic variables for each thread. It is as slow as global memory and not cached. Registers, the fastest memories, are also for automatic variables of each thread, but the number of 32-bit registers is limited.

2.3. Parallelizaiton of IMGA for UA-FLP on a GPU

There are only a few projects studying how to parallelize IMGA on GPUs. N. Melab and E. G. Talbi [33] proposed three different schemes to build efficient IMGA on GPUs. According to the experimental results, GPU-based IMGA can accelerate the search process speed, especially for large-scale optimization problems. Steffen and Dietmar [34] studied two models for evolutionary algorithms, including island model genetic algorithm, and discussed the implementation and performance under different parallel architectures such as CPU clusters and GPUs. Cheng-Chieh Li et al. [39] took the advantage of GPUs to parallelize IMGA, where they replace the mutation operator by simulated annealing when solving the travelling salesman problem (TSP).

In the latest literature, we first proposed how to use GPUs to parallelize the IMGA for solving UA-FLP [32]. The structure of our parallel IMGA on a GPU is illustrated in Figure 5. There are N islands, organized as a ring topology, and on each of them there are n individuals. Each individual of the population is represented with a chromosome that corresponds with a facility layout. Each individual has a fitness value that represents the quality of the solution, where the fitness value is calculated by Equation (1). FBS is used to encode a chromosome, consisting of a facility sequence and a bay division. Each island is assigned to one block on a GPU. The information about all individuals is stored in global memory. Each thread performs the procedures of IMGA, including initialization, evaluation, selection, crossover, mutation, and migration. Regarding the selection, important data are stored in shared memory in order to increase the access speed. In addition, migration is carried out through global memory, where threads in different blocks can communicate with each other efficiently using memory coalescing.

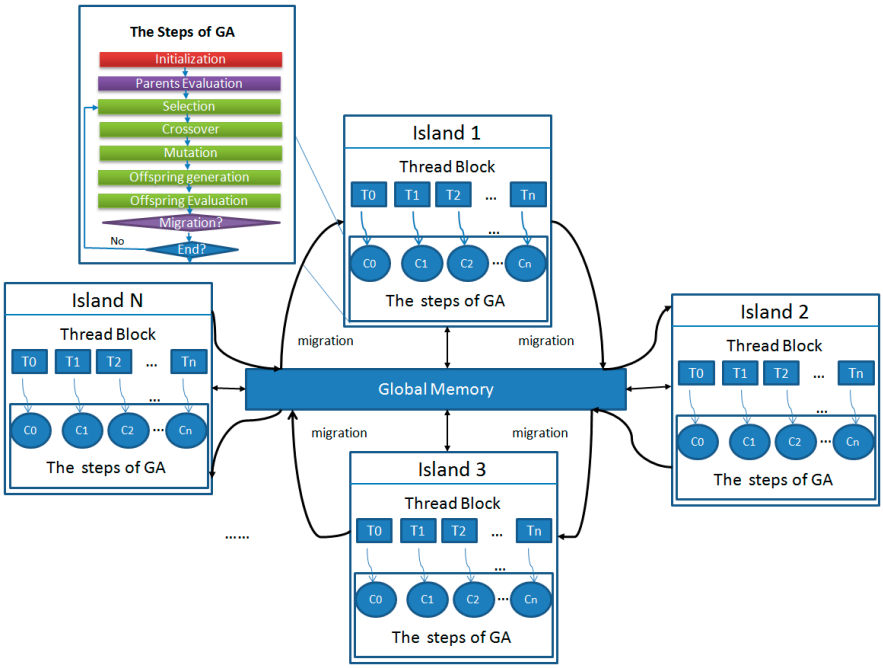

Figure 5. The main implementation structure of our GPU-accelerated IMGA for UA-FLP.

In our previous implementation, we proposed various parallel algorithms for each step of IMGA when solving UA-FLP. Experiments were conducted to compare the performances between different parallel versions and the CPU version. According to the experimental results, the performance improvement ratio of applying the basic parallelization methods for main steps of IMGA over the CPU

version is more than 20 at best. Furthermore, applying our suggested parallel methods to implement IMGA on a GPU, the best performance improvement ratio over the CPU version can be as high as 84. That is, our GPU version provides much higher performance than both the conventional GPU version and the CPU version.

The previous literatures discuss how to use a GPU to accelerate IMGA in order to reduce the execution time, and thus improve the system performance. Since both CPU and GPU versions adopt the same IMGA algorithm, the solutions will be the same. Consequently, solution quality was not usually discussed in the reports. Nevertheless, when executing IMGA on a GPU, different parameter settings will result in different solution qualities and execution times. Especially for large-scale optimization problems, adjusting parameter values usually leads to a much longer execution time, so even if the solution quality will improve or not, is uncertain. In addition, each parameter has its own impact on solution quality and execution time. For some parameters, increasing its value will improve the best solution a little, but the execution time might grow exponentially. By contrast, increasing another parameter value might improve the best solution significantly, but the execution time only becomes longer polynomially. Therefore, if a user wants to find a better solution on a GPU, but they do not want a longer execution time, it is crucial to give them a guideline on how to adjust individual parameters. Instead of only performance tuning, we focus in this paper on how to adjust parameters efficiently, in order to have a better solution at a lower cost of increment of execution time when solving UA-FLP with IMGA on a GPU.

In IMGA, selection is one of the most time-consuming and quality-influencing operations. The parallelisms of the two most common selection methods on a GPU have been discussed in our previous work [32]. In the following sections, we firstly proposed an effective method of generating random seeds for the tournament method on a GPU, which can ensure us how to find a better quality of solution, within a shorter execution time. We then focused on how to tune the relevant parameters to have a better solution quality, after trading off between solution quality and execution time when GPUs are used to accelerate IMGA. The results will help researchers set parameters in a more efficient and effective way to ensure better quality solutions, without increasing execution time too much.

3. Improvement of Parallel Tournament Selection

In the tournament selection, each iteration has pairs of individuals from the population, which have to be randomly selected for fitness value comparisons. The winning individuals are determined after a series of selections and comparisons. Random numbers can be generated with the CUDA library and a random seed is required to initialize the pseudorandom number generator. The execution time of the tournament method on a GPU varies depending on how the random seeds are generated. In addition, the random seed generation method also influences the final solution quality. Three methods of random seed generation are proposed and discussed in this section. The random seed can be generated by the following methods. (1) The system clock is used to generate only one random seed. (2) The system clock is used to generate one random seed for each comparison. (3) Thread IDs are used to generate only one random seed. Experiments are conducted in order to compare these methods.

3.1. Generate Random Seed Once with the System Clock

In this method, the system clock is used to generate a random seed. Since the random seed is generated only once during the whole tournament selection, there is only one set of random numbers for every thread to perform selection and comparison. In order to get two different positions to compare, threads use their IDs to have different starting offset. As a result, the second random number acquired by the first thread will be the same as the first random number acquired by the second thread. This method will significantly narrow down the search space explored during the tournament, resulting in a worse solution quality of IMGA.

The example is shown in Figure 6, assuming that there is a four-round tournament selection, and the number of thread is four. Thread 1, 2, 3 and 4 will conduct tournament selection separately

on the sets of (9, 121, 32, 18), (121, 32, 18, 1) and (32, 18, 1, 136), (18, 1, 136, 6). Among the four rounds of selection, there are 9 repeated comparisons. If the number of rounds in a tournament is M, which are conducted by T threads, there will be $\frac{T(T-1)}{2} + (M-1)(T-1)$ non-repeated comparisons and $\frac{(T-2)(T-1)(M-1)}{2}$ repeated comparisons. Obviously, the larger the values of M and T are, the more the number of repeated comparisons, resulting in a poor quality of solution. The kernel Algorithm 1 shows the pseudocode of generating the random seed with the system clock only once to parallelize the tournament selection.

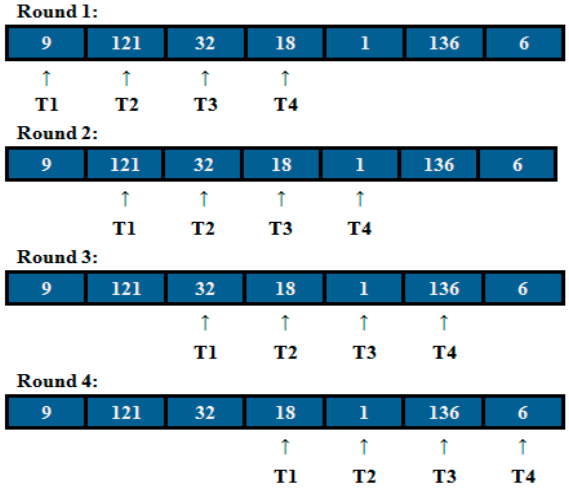

Figure 6. The example of generating the random seed once by the system clock.

Algorithm 1: Generate Random Seed Once

```
Data:
  round: number of tournaments that are conducted
  population: number of chromosomes that are in an island
Input:
  survive; fitness
Function:
  clock(): access the system time from GPU
  rand_initiate(): initiate the state of random number generator
  rand_generate(): generate a random number
Result:
  Chromosomes are randomly picked, and conduct a series of tournament.
Parallel:
1:  tseed = clock()
2:  offset = threadId
3:  rand_initiate(tseed, offset, state)
4:  winner = 0
5:  for i = 1 to i = round do
6:      rival = rand_generate(state) mod population
7:      if fitness[blockId][rival] < fitness[blockId][winner] do
8:          winner = rival
9:      end
10: end
11: survive[blockId][threadId] = winner
```

3.2. Generate a Random Seed during Each Round of Comparisons

Because generating the random seed only once can explore a relatively smaller search space in the tournament selection, we can generate random seeds during each round of comparisons to explore

a larger search space. There is an example shown in Figure 7. In the same round of comparisons, each thread will have the same random seed, but each round of comparison has its random seed, resulting in a new sequence of random numbers for each round of comparison. This way, each thread has different random numbers in each round of the comparisons for the tournament selection. For example, Thread 1 will get the random number set of (9, 8, 68), Thread 2 gets (121, 16, 47), Thread 3 gets (32, 91, 76), Thread 4 gets (18, 0, 23), and Thread 5 gets (1, 54, 92). Algorithm 2 shows the pseudocode of generating random seeds for each round of comparison. With this strategy, there are almost no repeated comparisons, but this method will bring additional cost of the execution time caused by regenerating random seeds.

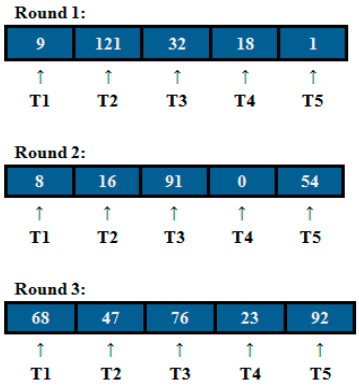

Figure 7. The example of generating a random seed during each round of comparison.

Algorithm 2: Generate a Random Seed during Each Round of Comparison

```
Data:
  round: number of tournaments that are conducted
  population: number of chromosomes that are in an island
Input:
  survive; fitness
Function:
  clock(): access system time from GPU
  rand_initiate(): initiate the state of random number generator
  rand_generate(): generate a random number using the state of generator
Result:
  Chromosomes are randomly picked, and conduct a series of tournament.
Parallel:
1:  offset = threadId
2:  winner = 0
3:  for i = 1 to i = round do
4:      tseed = clock()
5:      rand_initiate(tseed, offset, state)
6:      rival = rand_generate(state) mod population
7:      if fitness[blockId][rival] < fitness[blockId][winner] do
8:          winner = rival
9:      end
10: end
11: survive[blockId][threadId] = winner
```

3.3. Our Improved Strategy

In order to avoid repeated comparisons resulting in the worse quality of solution, and to avoid spending much time on updating random seeds for each round of comparisons, we propose a new strategy to implement the parallel tournament selection. At first, a group of random seeds are generated

at the same time with parameters including thread ID, block ID and the system clock. Therefore, each round of comparison will have a different random seed to generate a different sequence of random numbers. Then, the threads perform the comparisons based on the random numbers during each round. Adopting this method, threads can obtain different sequences of random numbers, without regenerating the random seed. For example, as shown in Figure 8, *M* random seeds are randomly generated by each thread with its thread ID, block ID and the system clock. Based on the *M* random seeds, *M* sequences of random numbers are produced for *M* rounds. Using this method, the random seeds are generated only once, but there are no repeated comparisons for most of the time. The solution quality provided by our improved strategy is much better than that of the method that uses the system clock to generate the random seed only once. In addition, our solution quality is almost the same as that produced by the method that generates one random seed for each round of comparisons, but our method can significantly reduce the execution time caused by frequently updating random seeds. Algorithm 3 shows the pseudocode of our improved strategy.

Round / THD#	R1	R2	R3	R4	R5	R6	.....	RM
THD 1	5	16	9	72	81	43	.....	62
THD 2	85	91	40	0	21	69	.....	25
.....	.....							
THD N	1	12	84	36	87	21	41	51

Figure 8. Our improved strategy for the parallel tournament selection.

Algorithm 3: Our Improved Strategy

```
Data:
  round: number of tournaments that are conducted
  population: number of chromosomes that are in an island
Input:
  survive; fitness
Function:
  clock(): access system time from GPU
  rand_initiate(): initiate the state of random number generator
  rand_generate(): generate a random number using the state of generator
Result:
  Chromosomes are randomly picked, and conduct a series of tournament.
Parallel:
1:  tseed = clock() + blockId * population + threadId
2:  offset = 0
3:  rand_initiate(tseed, offset, state)
4:  winner = 0
5:  for i = 1 to i = round do
6:      rival = rand_generate(state) mod population
7:      if fitness[blockId][rival] < fitness[blockId][winner] do
8:          winner = rival
9:      end
10: end
11: survive[blockId][threadId] = winner
```

4. Performance Evaluations

4.1. Tested Platform

We adopted NVIDIA GeForce GTX 980 to evaluate our GPU version of IMGA for UA-FLP. The GTX980 is featuring Maxwell microarchitecture, which can improve graphics capabilities, simplify programming, and have better energy efficiency. GTX980 has 2048 CUDA cores distributed over 16 SMs with a clock frequency of 1126 MHz [40]. The CUDA cores can access to global memory with a size of 4096 Mbytes and memory bus width of 256-bit. Total amount of shared memory per block is 49,152 bytes, and total number of registers available per block is 65,536.

The workstation mainly consists of one Intel Xeon CPU E5-2609 v3 with 16 GB memory and one GTX980 with 4 GB memory. Detailed configurations are shown in Table 1.

Table 1. Configuration of our 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

According to our previous work [32], by increasing the number of facilities, the GPU gains larger speed-up compared to CPU, the same conclusions are also reported in related paper [36], meaning GPUs can provide very high performance improvements for solving large-scale facility layout problems. Therefore, we select the larger data set of facilities for the experiments. The problem data sets are shown in Table 2, which illustrates the detailed information about data sets including name, number of facilities, facility size, aspect ratio, and distance measure.

Table 2. The problem data sets.

No.	**Problem Data Set**	**Number of Facilities**	**Facility Size**		**Common Shape Constraint**	**Distance Measure**
			Width	**Length**		
1	F40	40	45.00	45.00	α = 1000	Rectilinear
2	F60	60	45.00	45.00	α = 1000	Rectilinear
3	F80	80	45.00	45.00	α = 1000	Rectilinear
4	F100	100	45.00	45.00	α = 1000	Rectilinear
5	F120	120	45.00	45.00	α = 1000	Rectilinear

In the IMGA, the three most important parameters that have a high impact on both solution quality and execution time, are the number of islands, the number of generations (the number of iterations), and the subpopulation size per island (the number of chromosomes per island). Each one of the parameters have a different effectiveness on improving solution quality and impacts on lengthening execution time. We will analyze every parameter via a series of experiments to understand how much the solution quality can be improved at the cost of increasing an execution time unit. Based on the analysis result, we could give users a guideline on the selection ordering of parameters for tuning the solution quality and execution time. Moreover, we will find a parameter setting that can obtain a better solution without increasing the execution time significantly when using a GPU to implement IMGA for solving UA-FLP.

According to the hardware features of the GPU, the settings of the IMGA parameters on the GPU in our evaluation are listed in Table 3. In accordance with the previous work [20], the baseline of the parameter settings of the IMGA is: 16 islands, 128 iterations, and 128 chromosomes per island. The reason why we choose 2^k as the value of each parameter is mainly dependent on the hardware characteristics of the GPU.

Table 3. Settings of the IMGA parameters on GPU in our evaluations.

Parameter	Value
Total population size per island (n)	32, 64, 128, 256, 512,1024
Number of islands (N)	16, 32, 64, 128, 256
Cycle generations (c)	64, 128, 256, 512, 1024, 2048, 4096, 8192
Migration rate (m)	5
Migration frequency (g)	15
Crossover probability (p_c)	0.7
Mutation probability (p_m)	0.01

4.2. Evalutions of Parallel Tournament Selection

In Section 3, we have introduced three methods of generating random seeds in the tournament selection. The first method generates a random seed only once using the system clock, and the second method is to generate a random seed for each round of comparisons. In the third method, each thread generates different random seeds based on its thread ID, block ID and the system clock at one time.

To evaluate the three methods above three, we use data sets with different numbers of facilities, as shown in Table 2. The values of three main parameters for IMGA are set as follows. The number of chromosomes is 128, the number of iterations is 128, and the number of islands is 16. Each data set is tested five times to calculate the average execution time and to have the best solution. Method 1 is adopted as the base of comparison. We use Quality Improvement Ratio and Normalized Execution Time to compare the three methods, which are defined by Equation (5) and Equation (6):

$$\text{Quality Improvement Ratio} = \frac{\text{Solution Quality of Method 1}}{\text{Solution Quality of Method X}} \ (\text{X} = 2,\ 3) \tag{5}$$

$$\text{Normalized Execution Time} = \frac{\text{Execution Time of Method X}}{\text{Execution Time of Method 1}} \ (\text{X} = 2,\ 3) \tag{6}$$

The comparison of Quality Improvement Ratio is illustrated in Figure 9. According to Figure 9, the quality obtained by Method 2 or Method 3 is increased by about 10% compared to that by Method 1. The qualities provided by Methods 2 and 3 are almost the same for all cases, which implies that our improved method can guarantee the obtainment of a better-quality solution. Figure 10 shows the comparison of Normalized Execution Time. The execution time taken by Method 3 is the shortest one, and almost the same as that of Method 1. Method 2 takes about 1.5 to 4 times longer execution time than that of Method 1, and performance degradation becomes worse as the number of facilities decreases. This indicates that the smaller the number of facilities, the more execution time the selection operation takes. In short, according to the experimental results our improved method can provide the best-quality solution and take the shortest execution time, which provides the best implementation of the tournament selection on a GPU.

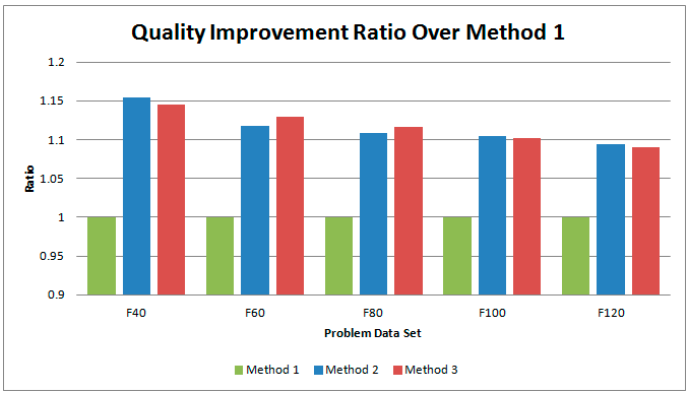

Figure 9. Comparison of the quality improvement ratio over Method 1.

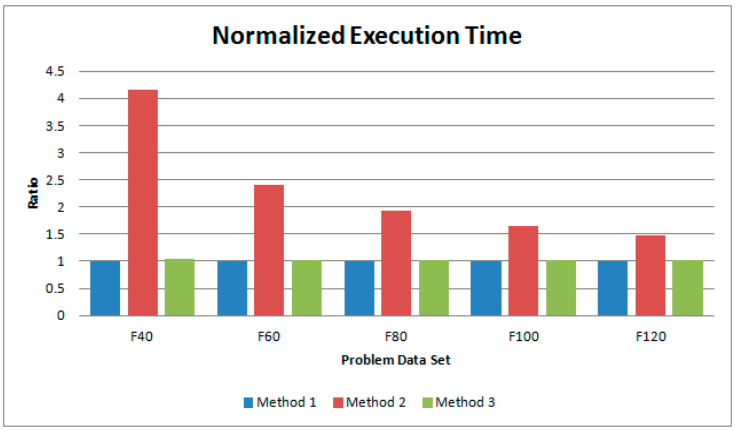

Figure 10. Comparison of the normalized execution time.

4.3. Evaluations of Quality Improvement

This part of the experiments, focuses on how to use IMGA on GPU to achieve better quality within a shorter amount of time, taking UA-FLP as an example. In the experiment, we tuned three parameters: The number of islands (*ns*), the number of iterations (*ni*), and the number of chromosomes (*nc*), to discuss the solution quality and the execution time to find a combination of good parameter setting. The parameters of the CPU version are set to the values used in the literature [18,41] and they are: 16 islands, 128 iterations, and 128 chromosomes per island, which will be the baseline version for comparison. Each of the benchmarks, listed in Table 2, are executed ten times to have the average execution time and the best-quality solution. In the following evaluations, solution quality and execution time are normalized to the baseline version. We define three kinds of criteria to discuss how each of the three parameters influences solution quality and execution time. They are: Normalized solution quality (NQ), normalized execution time (NT), and normalized solution quality/execution time ratio (QTR). The three formulas are defined by Equations (7)–(9), as shown below:

$$\text{NQ} = \frac{\text{solution quality on CPU}}{\text{solution quality on GPU}} \tag{7}$$

$$\text{NT} = \frac{\text{execution time on GPU}}{\text{execution time on CPU}} \tag{8}$$

$$QTR = \log \frac{NQ}{NT} \tag{9}$$

NQ indicates how many percentages the solution quality a GPU version can provide, and NT indicates how many percentages the execution time a GPU version will increase. QTR is the ratio of NQ and NT, indicating how much the cost of execution time increment will be for a better solution. A larger QTR implies a more cost-effective GPU version. In the following three subsections, we will use the above criteria to analyze three key parameters: the number of islands, the number of chromosomes per island, and the number of iterations.

4.3.1. Influence of the Number of Islands

Firstly, we only tune the number of islands, and the rest of the parameters are unchanged. That is, the number of chromosomes per island and the number of iterations are both set to 128, but the number of islands per island varies in the range of 16, 32, 64, 128 and 256. We discuss how the parameter value of the number of islands influences solution quality and execution time. The NQ, NT and QTR are shown in Figures 11–13. It can be concluded that as the number of islands increases, the longer execution time is required. When the number of islands is less than or equal to 128, the quality of solution is improving when the number of islands increases, as shown in Figure 11. However,

when the number of islands is greater than 128, the quality becomes worse, as shown in Figure 11, and the execution time becomes even longer, as shown in Figure 12. Moreover, the values of the QTR become smaller and smaller when there are more islands, especially when we use more than 64 islands, as shown in Figure 13, indicating that it is not cost-effective to pursue a high-quality solution using more than 64 island. In summary, the best case is 16 islands.

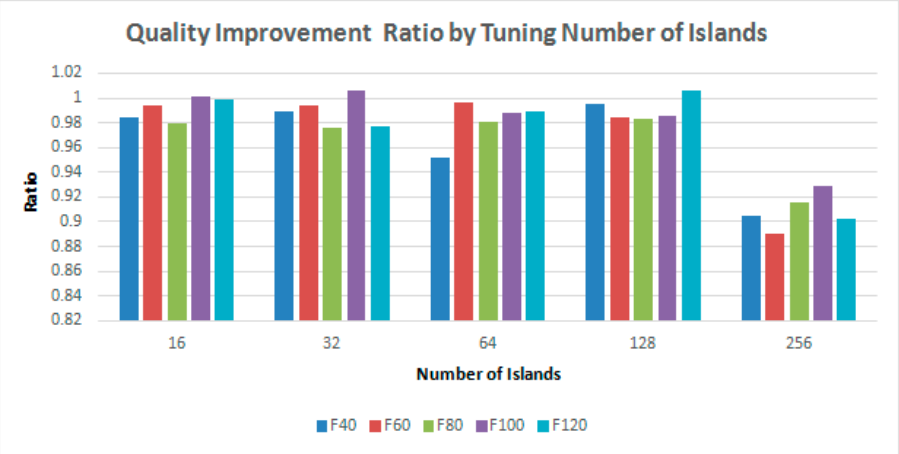

Figure 11. Comparison of NQ values for different numbers of islands.

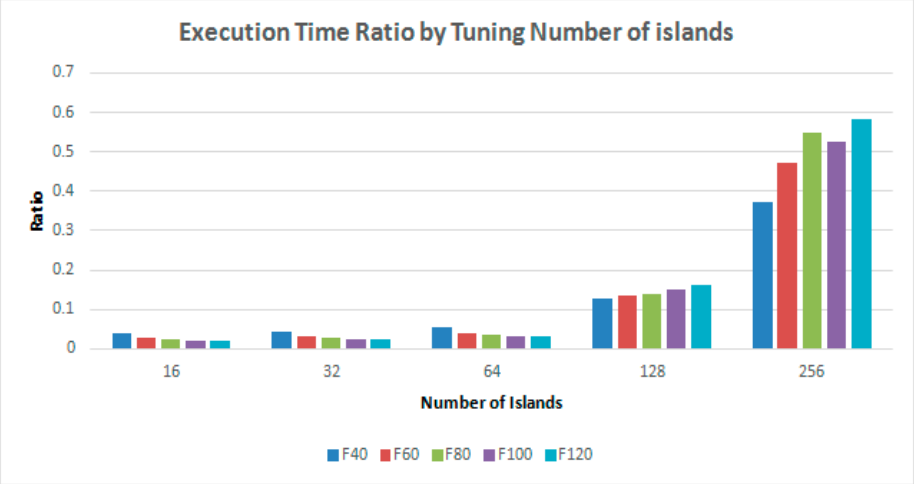

Figure 12. Comparison of NT values for different numbers of islands.

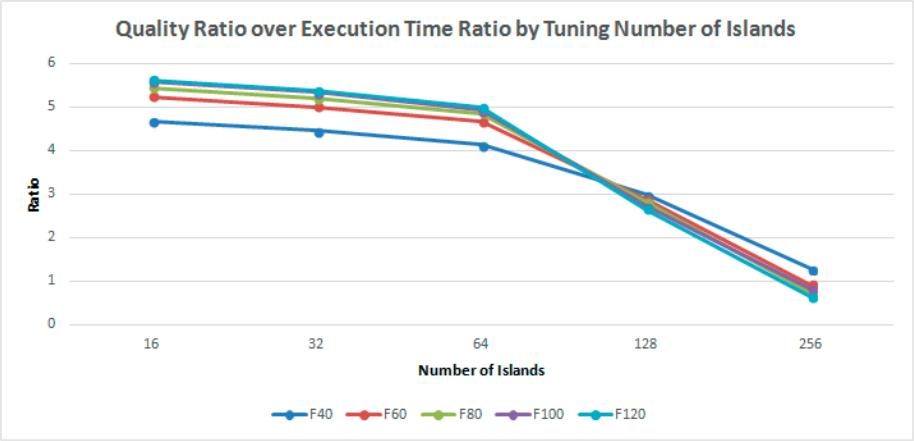

Figure 13. Comparison of QTR values for different numbers of islands.

4.3.2. Influence of the Number of Iterations

In this part, the number of islands and the number of chromosomes are set to 16 and 128, respectively. The values of the number of iterations to be evaluated include 64, 128, 256, 512, 1024, 2048, 4096 and 8192. The results of the experiments for NT, NQ and QTR are shown in Figures 14–16. The quality is improved gradually when the number of iterations is enlarged, as shown in Figure 14. Basically, when the number of iterations reaches 128, the improvement of solution quality is similar to that provided by CPU. On the other hand, the speedup of the execution time obtained by GPU over CPU becomes much larger when we increase the problem size, as shown in Figure 15. The same conclusion is consistent to that reported in the related work [36]. In general, the execution time is lengthened linearly with the increased number of iterations. According to Figure 16, the value of QTR decreases linearly, as the number of iterations increases. In particular, when there are more than 2048 iterations, the QTR becomes negative, indicating the performance is worse than the CPU version. In accordance with the experimental results, when the number of iteration is equal to or larger than 128, a better performance can be achieved.

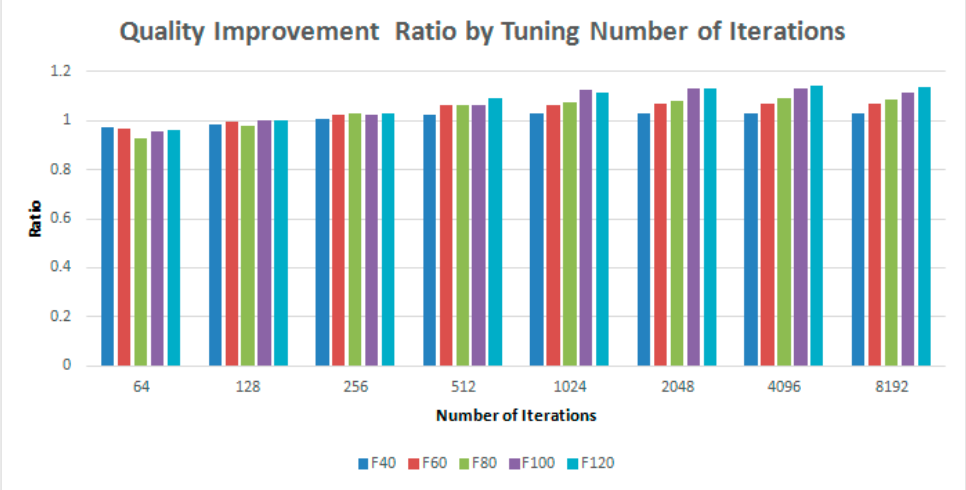

Figure 14. Comparison of NQ for various numbers of iterations.

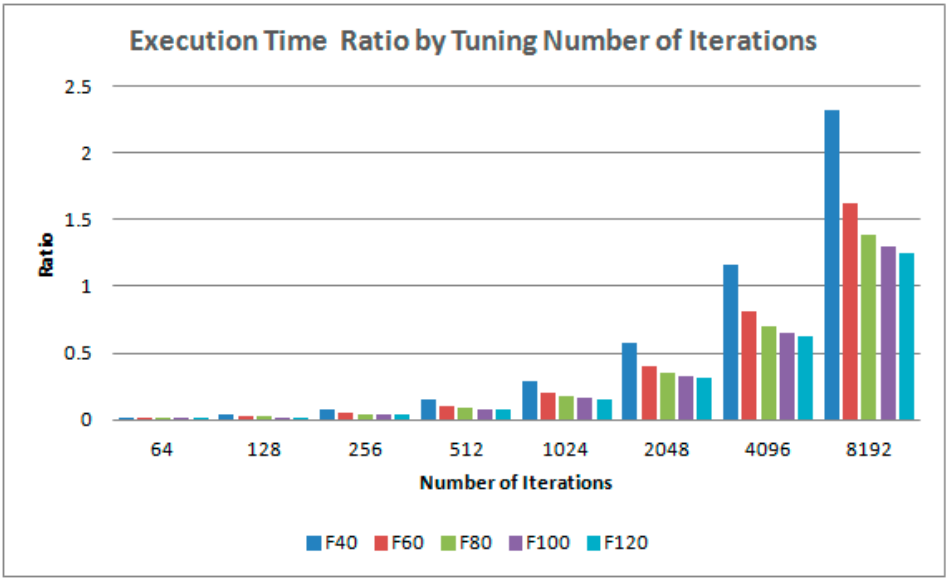

Figure 15. Comparison of NT for different numbers of iterations.

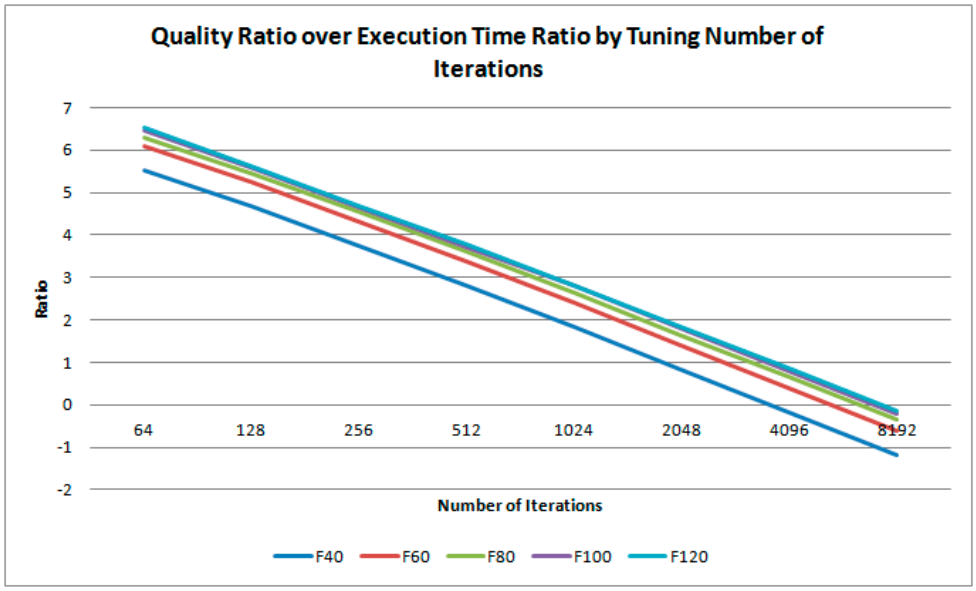

Figure 16. Comparison of QTR for different numbers of iterations.

4.3.3. Influence of the Number of Chromosomes Per Island

At last, we tune the number of chromosomes per island in the set of (32, 64, 128, 256, 512, 1024) while fixing the number of islands and iterations to 16 and 128, respectively. Comparisons of NT, NQ and QTR are shown in Figures 17–19. When the number of chromosomes per island becomes larger, the solution quality gets better and better, as shown in Figure 17. However, the solution quality can be better than the CPU baseline, only when the number of chromosomes per island is equal to, or more than 128. The larger the problem size, the smaller the increment of the execution time, as shown in Figure 18, meaning that it is more worthwhile for problems with a large number of facilities to increase the number of chromosomes per island in order to have a better solution. As for QTR as shown in Figure 19, after the number of chromosomes per island increases to 256, the value will drop down significantly. However, in general the values of QTR are all larger than one, indicating that increasing the number of chromosomes per island on a GPU can usually have better solutions than the CPU baseline. In addition, to pursue a better solution, it is better to increase the number of chromosomes per island than to increase the number of iterations, as shown in Figure 13.

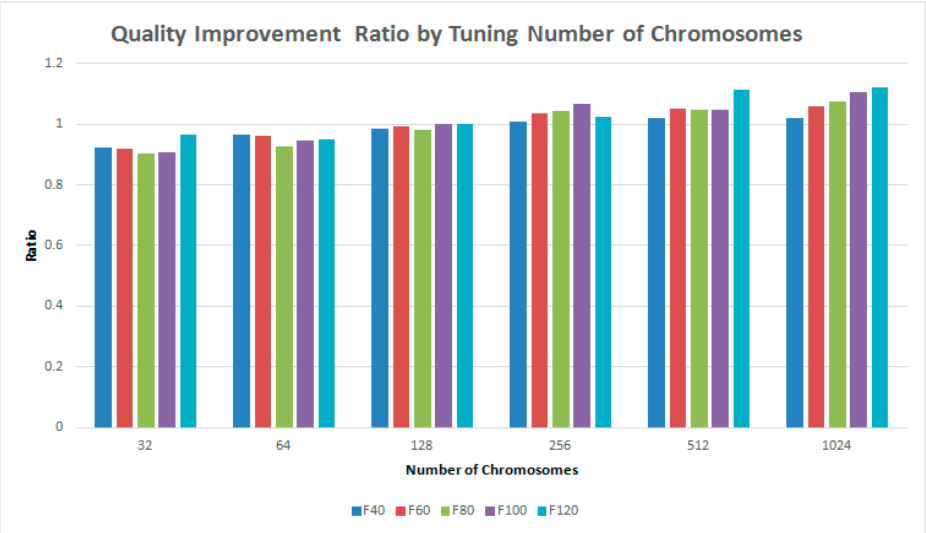

Figure 17. Comparison of NQ for different numbers of chromosomes per island.

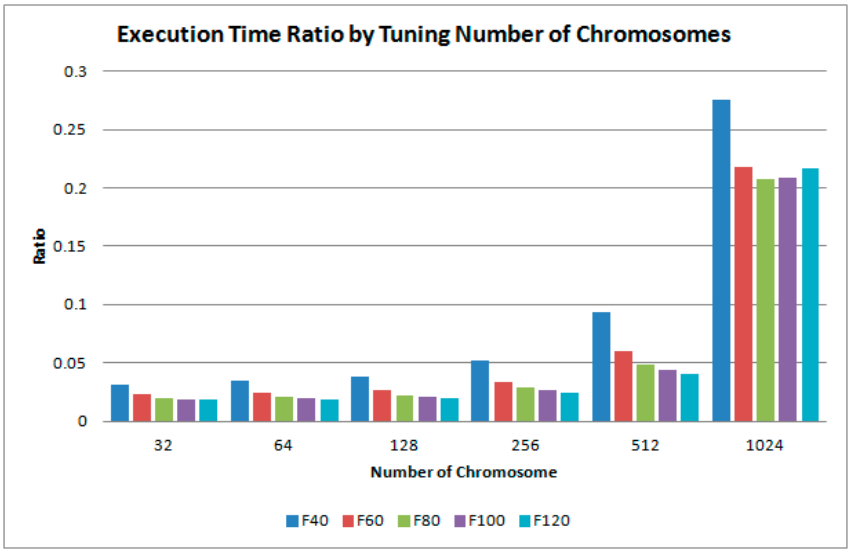

Figure 18. Comparison of NT for different numbers of chromosomes per island.

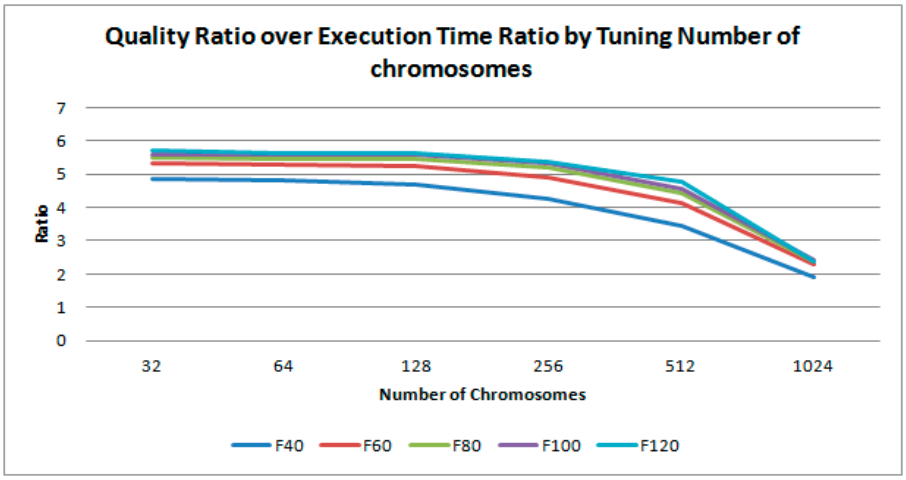

Figure 19. Comparison of QTR for different numbers of chromosomes per island.

4.3.4. Effect of Combining Suggested Parameter Settings

According to the above experiments, we suggest the following combination of parameter settings when using the GPU to execute IMGA: The number of islands is 16, the number of iterations is 128, and the number of chromosomes per island is 256. With these suggested parameter settings, the values of NQ, NT and QTR for each benchmark are shown in Table 4.

Table 4. NQ, NT and QTR for each benchmark with recommended parameter settings.

Benchmark	NQ	NT	QTR
40	1.0097	0.2390	2.0788
60	1.0266	0.0752	3.7712
80	1.0020	0.0518	4.2739
100	1.0332	0.0410	4.6543
120	1.0316	0.0350	4.8822

All values of NQ are greater than 1, as shown in Table 4, meaning that the solution quality after using our recommended parameter settings, GPU always can provide better solutions CPU for each benchmark. Comparing with the CPU baseline version, by using the GPU version with our proposed

parameter settings, the quality of solution can be improved by 3%, and the execution speed is 29 times faster than the CPU version. Meanwhile, as the number of facilities increases, the value of QTR also increases gradually. This indicates that the larger the number of facilities, the higher the cost effective to have a better solution using a GPU. That is, compared with the CPU counterpart, the GPU can not only improve the solution quality, but also shorten the execution time significantly.

In addition, we have conducted another experiment in order to realize how much the improvement of solution quality can be provided by GPU, when GPU and CPU are given almost the same execution time. We have conducted many experiments, and found that the execution times of GPU and CPU are very close when the number of chromosomes per islands, the number of iterations, and the number of islands are 128, 4096 and 16, respectively. Under the above parameter setting, the quality of solution can be greatly improved by the GPU version. We use the percentage of quality improvement (PQI) to represent how much the improvement of the solution quality the GPU version can provide, and it is defined in Equation (10) below, where QCPU and QGPU are the best solution qualities that CPU and GPU can provide, respectively. Figure 18 shows the PQI for the data sets. As shown in Figure 20, the solution qualities of all the benchmarks are significantly improved. As the number of facilities increases, the quality gets better. When the number of facilities is 120, the quality improvement based on PQI can be up to 8%.

$$\text{PQI} = \frac{QCPU - QGPU}{QCPU} \tag{10}$$

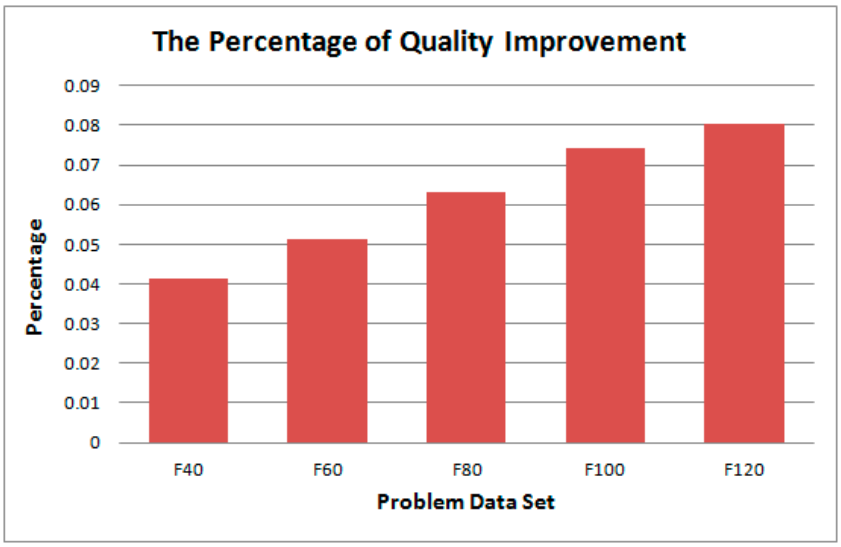

Figure 20. Comparison of the percentage of quality improvement.

5. Conclusions

The island model is a popular and efficient method to implement the genetic algorithm on a parallel architecture. In recent work, the IMGA is applied to solve UA-FLP, which can improve the solution quality for most of the problem data sets. However, the amount of calculation will become larger and larger with the increase in the size of the problem, because UA-FLP is an NP-hard problem. Moreover, the execution time is getting longer and longer even though metaheuristic approaches are used to solve UA-FLP. Modern GPUs, highly-parallel computing processors, can manipulate large amounts of data efficiently by embedded many cores. So far, almost all research on using GPUs to execute IMGA investigates how to reduce the execution time. For the researchers who work on practical problems, finding a solution of higher quality is definitely the key concern.

In this paper, we study how to achieve better qualities in a more cost-effective way when using GPU to accelerate IMGA for solving large-scale problems. We take UA-FLP as an example in this study.

Firstly, we addressed the problem of how random seed generation on GPU influences the solution quality. Random seeds can be generated based on two different methods: (1) Generate only one seed with the system clock, or (2) generate one seed with the system clock for each round of comparisons. The first method suffers from the problem of repeated comparisons, which limits the search space explored significantly. The second method can provide a better solution at the cost of much longer execution time. Therefore, we proposed an efficient method to generate only one random seed with the multiple parameters: Thread ID, block ID and the system clock. Consequently, each thread can have a unique sequence of random numbers in each round. According to the experimental results, our method can provide a solution as good as that provided by the first method, and an execution time as short as that of the second method.

Next, we addressed the challenge of how to trade off between solution quality and execution time when we set parameters. Three important parameters of IMGA are investigated, including the number of islands, the number of generations, and the number of chromosomes per island. According to the experimental results, to have a better solution at the low cost of execution time, the order of influence on solution quality is: The number of chromosomes per island, the number of generations, and then the number of islands. Moreover, when executing IMGA on a GPU for solving UA-FLP, we also recommend a set of parameter settings to find a more cost-effective solution based on a series of experiments. The recommended values of the three parameters are: 16 islands, 128 generations, and 256 chromosomes per island. With the above parameter setting, the quality of solution on GPU is improved by about 3% over the CPU baseline version. In addition, the GPU version is 29 times faster than the CPU baseline. Furthermore, if we let GPU and CPU spend almost the same execution time, the quality improvement by GPU can be up to 8%.

In the future, we will further discuss how to improve solution qualities of different NP-hard problems on GPUs when IMGA is adopted. We believe that our parallelization strategy can provide an elegant and cost effective way of solving these problems on a desktop computer simply equipped with a graphic card, instead of relying on the large computational grid.

Author Contributions: Conceptualization, X.S., 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., and C.-C.W.; Investigation, X.S., P.C., and C.-C.W.; Methodology, X.S., P.C., and C.-C.W.; Project administration, 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., C.-C.W., and L.-R.C.; Writing—review & editing, X.S., C.-C.W., and L.-R.C.

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

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

References

1. 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]
2. 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]
3. Wong, K.Y. Solving facility layout problems using Flexible Bay Structure representation and Ant System algorithm. *Expert Syst. Appl.* **2010**, *37*, 5523–5527. [CrossRef]
4. Kulturel-Konak, S.; Konak, A. Unequal area flexible bay facility layout using ant colony optimisation. *Int. J. Prod. Res.* **2011**, *49*, 1877–1902. [CrossRef]
5. 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. Manage.* **2016**, *11*, 259. [CrossRef]
6. 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.
7. Kenneth, A.D.J. An analysis of the behavior of a class of genetic adaptive systems. Ph.D. Thesis, University of Michigan, Ann Arbor, MI, USA, 1975.

8. 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.
9. Khuda Bux, N.; Lu, M.; Wang, J.; Hussain, S.; Aljeroudi, Y. Efficient association rules hiding using genetic algorithms. *Symmetry* **2018**, *10*, 576. [CrossRef]
10. Goldberg, D.E.; Holland, J.H. Genetic algorithms and machine learning. *Mach. Learn.* **1989**, *3*, 95–99. [CrossRef]
11. Ijjina, E.P.; Chalavadi, K.M. Human action recognition using genetic algorithms and convolutional neural networks. *Pattern Recognit.* **2016**, *59*, 199–212. [CrossRef]
12. 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]
13. Shin, H.; Joo, C.; Koo, J. Optimal rehabilitation model for water pipeline systems with genetic algorithm. *Procedia Eng.* **2016**, *154*, 384–390. [CrossRef]
14. Deng, Q.; Gong, G.; Gong, X.; Zhang, L.; Liu, W.; Ren, Q. A bee evolutionary guiding nondominated sorting genetic Algorithm II for multiobjective flexible Job-shop scheduling. *Comput. Intell. Neurosci.* **2017**, *2017*. [CrossRef] [PubMed]
15. Alshamsi, A.; Diabat, A. a genetic algorithm for reverse logistics network design: A case study from the GCC. *J. Clean. Prod.* **2017**, *151*, 652–669. [CrossRef]
16. RazaviAlavi, S.; AbouRizk, S. Site layout and construction plan optimization using an integrated genetic algorithm simulation framework. *J. Comput. Civil Eng.* **2017**, *31*, 04017011. [CrossRef]
17. Szénási, S.; Vámossy, Z. Implementation of a distributed genetic algorithm for parameter optimization in a cell nuclei detection project. *Acta Polytech. Hung.* **2013**, *10*, 59–86.
18. Szénási, S.; Felde, I. Configuring genetic algorithm to solve the inverse heat conduction problem. In Proceedings of the 2017 IEEE 15th International Symposium on Applied Machine Intelligence and Informatics (SAMI 2017), Herlany, Slovakia, 26–28 January 2017.
19. Pospichal, P.; Jaros, J.; Schwarz, J. Parallel Genetic Algorithm on the CUDA Architecture. In *Applications of Evolutionary Computation*; Springer: Berlin, Germany, 2010; pp. 442–451.
20. 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]
21. Starkweather, T.; Mcdaniel, S.; Whitley, D.; Mathias, K.; Whitley, D.; Dept, M.E. A comparison of genetic sequencing operators. In Proceedings of the Fourth International Conference on Genetic Algorithms, San Diego, CA, USA, 7–11 July 1991; pp. 69–76.
22. Meller, R.D.; Gau, K.Y. The facility layout problem: Recent and emerging trends and perspectives. *J. Manuf. Syst.* **1996**, *15*, 351–366. [CrossRef]
23. Tompkins, J.A.; White, J.A.; Bozer, Y.A.; Tanchoco, J.M.A. *Facilities Planning*; John Wiley &Sons: New York, NY, USA, 2010.
24. Garey, M.R.; Johnson, D.S.; Stockmeyer, L. Some simplified NP-complete graph problems. *Theor. Comput. Sci.* **1976**, *1*, 237–267. [CrossRef]
25. Hasda, R.K.; Bhattacharjya, R.K.; Bennis, F. Modified genetic algorithms for solving facility layout problems. *Int. J. Interact. Des. Manuf.* **2017**, *11*, 713–725. [CrossRef]
26. Leno, I.J.; Sankar, S.S.; Ponnambalam, S.G. An elitist strategy genetic algorithm using simulated annealing algorithm as local search for facility layout design. *Int. J. Adv. Manuf. Tech.* **2016**, *84*, 787–799.
27. Paes, F.G.; Pessoa, A.A.; Vidal, T. A hybrid genetic algorithm with decomposition phases for the unequal area facility layout problem. *Eur. J. Oper. Res.* **2017**, *256*, 742–756. [CrossRef]
28. Derakhshan, A.A.; Wong, K.Y.; Tiwari, M.K. Unequal-area stochastic facility layout problems: Solutions using improved covariance matrix adaptation evolution strategy, particle swarm optimisation, and genetic algorithm. *Int. J. Prod. Res.* **2016**, *54*, 799–823. [CrossRef]
29. García-Hernández, L.; Pierreval, H.; Salas-Morera, L.; Arauzo-Azofra, A. Handling qualitative aspects in unequal area facility layout problem: An interactive genetic algorithm. *Appl. Soft Comput.* **2013**, *13*, 1718–1727. [CrossRef]
30. Gonçalves, J.F.; Resende, M.G. A biased random-key genetic algorithm for the unequal area facility layout problem. *Eur. J. Oper. Res.* **2015**, *246*, 86–107. [CrossRef]
31. Szénási, S. Segmentation of colon tissue sample images using multiple graphics accelerators. *Comput. Biol. Med.* **2014**, *51*, 93–103. [CrossRef] [PubMed]

32. Sun, X.; Lai, L.F.; Chou, P.; Chen, L.R.; Wu, C.C. On GPU implementation of the island model genetic algorithm for solving the unequal area facility layout problem. *Appl. Sci.* **2018**, *8*, 1604. [CrossRef]
33. 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.
34. Limmer, S.; Fey, D. Comparison of common parallel architectures for the execution of the island model and the global parallelization of evolutionary algorithms. *Concur. Comput. Pract. Exp.* **2016**, *29*, e3797. [CrossRef]
35. Tong, X. SECOT: A sequential Construction Technique for Facility Design. Ph.D. Thesis, University of Pittsburg, Pittsburgh, PA, USA, 1991.
36. 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]
37. 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.
38. Jachym, M.; Lavernhe, S.; Euzenat, C.; Tournier, C. Effective NC machining simulation with OptiX ray tracing engine. *Vis. Comput.* **2018**, 1–8. [CrossRef]
39. 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*, 1687814017707413. [CrossRef]
40. NVIDIA. Whitepaper NVIDIA GeForce GTX 980. 2014. Available online: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF (accessed on 11 August 2018).
41. Whitley, D.; Rana, S.; Heckendorn, R. B. Island model genetic algorithms and linearly separable problems. In *AISB International Workshop on Evolutionary Computing*; Springer: Berlin, Germany, 1997.