A Lightweight Island Model for the Genetic Algorithm over GPGPU

Original Scientific Paper

Mohammad Alraslan
Idlib University, Faculty of Informatics Engineering, Department of Software Engineering
Idlib, Syria
Mohammad-Alraslan@idlib-university.com

Ahmad Hilal AlKurdi
Idlib University, Faculty of Informatics Engineering, Department of Software Engineering
Idlib, Syria
Ahmad-Hilal-AlKurdi@idlib-university.com

Abstract *– This paper presents a parallel approach of the genetic algorithm (GA) over the Graphical Processing Unit (GPU) to solve the Traveling Salesman Problem (TSP). Since the earlier studies did not focus on implementing the island model in a persistent way, this paper introduces an approach, named Lightweight Island Model (LIM), that aims to implement the concept of persistent threads in the island model of the genetic algorithm. For that, we present the implementation details to convert the traditional island model, which is separated into multiple kernels, into a computing paradigm based on a persistent kernel. Many synchronization techniques, including cooperative groups and implicit synchronization, are discussed to reduce the CPU-GPU interaction that existed in the traditional island model. A new parallelization strategy is presented for distributing the work among live threads during the selection and crossover steps. The GPU configurations that lead to the best possible performance are also determined. The introduced approach will be compared, in terms of speedup and solution quality, with the traditional island model (TIM) as well as with related works that concentrated on suggesting a lighter version of the master-slave model, including switching among kernels (SAK) and scheduled light kernel (SLK) approaches. The results show that the new approach can increase the speed-up to 27x over serial CPU, 4.5x over the traditional island model, and up to 1.5–2x over SAK and SLK approaches.*

Keywords*: GPGPU, Genetic algorithm, TSP, Island Model, Speed up*

1. INTRODUCTION

Nowadays, GPUs (Graphics Processing Units) are playing a significant role in general-purpose computing. Applications for engineering and research are accelerated by GPUs' capabilities in various scientific domains. Nvidia introduced CUDA (Computer-Unified Device Architecture) in 2007 as a general-purpose parallel computing API [1, 2]. Programmers can efficiently solve computational issues by utilizing the GPU's parallel architecture using CUDA. These days, laptops are equipped with powerful GPUs that have thousands of cores [3]. These reasons motivated the researchers to develop GPU-based parallel applications.

The traveling salesman problem (TSP) is a complex combinatorial optimization problem that has been used to solve many problems, like UAV path planning [4, 5]. A popular evolutionary algorithm for solving the TSP problem is the genetic algorithm (GA) [6]. The genetic algorithm is an iterative process that requires a lot of computation. To speed up the GA process, many studies have developed parallel approaches over the GPU. They relied on the basic models of the parallel genetic algorithm, including the master-slave model, the island model, and the cellular model. They developed them to suit the studied problem and decrease the execution time [7-9].

In the traditional master-slave model, each step of the genetic algorithm was mapped to a separate kernel. Performing the evolving process, in a single iteration, involves calling these kernels. There is much CPU-GPU communication overhead to call the kernels from the host in every iteration. The master-slave model based on persistent threads was suggested to collapse the multi-kernel into one kernel and keep the threads alive throughout the execution of all iterations in the genetic algorithm [10]. This approach reduces CPU-GPU communication because there is only one kernel call, and iterations are made within the kernel.

Calling a single kernel means that we will pass the number of participating threads (Grid and Block sizes) only once in order to execute the genetic algorithm. For this reason, a method was adopted to distribute the

work between these active and live threads to achieve the best possible performance.

In the traditional island model (TIM), the large population is divided into small subpopulations (islands). Each iteration is achieved with only one kernel call. This method also introduces a CPU-GPU communication overhead.

Converting this model to a method based on persistent threads requires discussing two issues: the synchronization mechanisms and the work distribution between live threads, whose number will be fixed throughout the kernel's execution.

In this paper, we focus particularly on how to convert the island model of the genetic algorithm into a new approach named the Lightweight Island Model (LIM) that follows the concept of persistent threads. For that, we highlight the earlier studies that worked on converting the traditional master-slave model into a lighter version by the number of kernel invocations. We will introduce two approaches and discuss the effects of the synchronization techniques used. The work distribution and the warp-based implementation will be presented at every step of the genetic algorithm. The introduced approach will be compared with the traditional island model and with similar and previous works.

The remainder of this paper is organized as follows: Section 2 introduces the previous works. Background information about the genetic algorithm for the TSP, GPU computing, and implementation details of the genetic algorithm over the GPU are highlighted in Section 3. Section 4 presents the lightweight island approaches, while Section 5 focuses on simulation and experimental results. Finally, we conclude the paper in Section 6.

2. PREVIOUS WORKS

The genetic algorithm is one of the most popular algorithms used to solve many complex problems, such as TSP. Many researchers address the acceleration of the genetic algorithm on GPUs to obtain results in a better time. Generally, researchers followed the three models of the parallel genetic algorithm: the master-slave model, the island model, and the cellular model.

Authors in [11, 12] introduced an approach based on the master-slave model. Each step of the genetic algorithm is associated with a separate kernel. This means a multi-kernel invocation and CPU-GPU data exchange at each iteration, which negatively affects performance.

An approach based on master-slave was also introduced in [13] to solve the traveling salesman problem. This approach depends on switching among kernels (SAK). It used three kernels to implement the steps in GA. The first kernel is responsible for fitting the population. The second kernel performs crossover, mutation, and fitness calculations. The third one executes the selection operator. The number of threads in the CUDA configuration was set equal to the number of individuals (grid-size*block-size=population-size). The time needed for kernel invocations is minimized because there is no data exchange between the host and the device. This method reduced the number of kernel invocations per iteration, but there are still a significant number of implicit synchronizations that can affect the execution time.

Many studies presented the island model of the GA over the GPU. A fully Distributed Island Model approach was introduced in [14]. In this approach, they ensured implicit global synchronization between the CPU and the GPU. This synchronization was performed by associating one kernel execution with one iteration of the evolutionary process. When the execution of one iteration is finished, the hand returns to the CPU. This synchronization mechanism decreases performance due to the large number of implicit synchronization points and the overhead of the kernels call. Each island was associated with a single block, and it contained 128 individuals (island-size). One individual was represented by one thread, which means that block-size=island-size= 128. The results are introduced for various numbers of islands (grid-size).

Authors in [15-18] also presented how to use GPUs to parallelize the island model of the genetic algorithm (IMGA). They focused on proposing parallel strategies for the genetic operators that are appropriate to the studied issue, but they did not address details about the mechanism for achieving global synchronization.

An approach named Scheduled Light Kernel (SLK) was presented in [10] for implementing GA on the GPU. This approach was inspired by the concept of persistent threads introduced in [19]. The introduced approach concentrated on persistently applying only the master-slave model to keep the threads alive inside a single kernel invocation. They collapse multiple kernel invocations into a single persistent kernel call. The execution method is determined by a work scheduling matrix. The GPU launching configuration (grid-size, block-size) of the persistent kernel is defined by finding the maximum number of blocks that is needed among the separate invocations. The researchers relied in their experiments on using the same number of available SMs (streaming multiprocessors) as a grid-size. As for the number of threads per block (block-size), they calculated the number of maximum threads as T = population-size (NPOP) / grid-size (B).

Since the aforementioned research did not focus, in its content, on developing a version of the persistent island model, we will provide the necessary details about that. We will make use of the research that applied the master-slave model using persistent threads to achieve the mechanism in the island model.

Considering that global synchronization greatly affects performance, we will discuss the mechanisms of synchronization and provide sufficient details about them. We will develop a warp-based parallelization strategy to achieve the best possible performance. We expect that the proposed approach will result in better performance.

3. BACKGROUND

3.1. THE GENETIC ALGORITHM FOR THE TSP PROBLEM

A genetic algorithm (GA) is an evolutionary problem-solving method based on Darwin's theory. This Algorithm evolves a group of solutions (population) by repeating some steps in sequence. GA starts by generating an initial population which is a group of solutions. After that, GA starts evolving these solutions in an iterative process. At each iteration, a set of steps are performed, including Crossover, Mutation, Evaluation, and Selection [20]. This process is repeated until it reaches a pre-specified condition. Fig. 1 shows the process of the GA.

The Traveling Salesman Problem (TSP) is one of the most studied combinatorial optimization problems. It is used in many real-world applications, such as UAV Path planning. The TSP should find the shortest route that visits a group of cities exactly once and returns to the initial city. The genetic algorithm is widely used to solve TSP and other NP-hard problems [6].

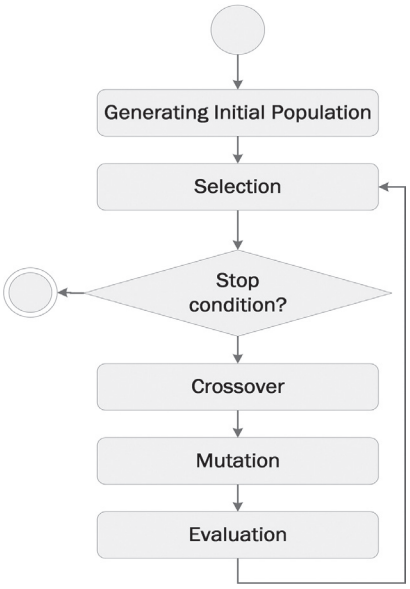

Fig. 1. Steps of the genetic algorithm

When using the genetic algorithm for solving TSP, each solution is represented by a route. A list of cities' indexes in a variation order makes up the route. The distances between cities are stored in an array. The fitness of each route is calculated by the distances array. There are various crossover techniques for the TSP, including PMX, CX, OX, etc.

3.2. GPU COMPUTING

In recent years, general-purpose GPUs (GPGPUs) have evolved into highly parallel, multithreaded, many-core processors with very high memory bandwidth [21]. Many laptops are now available with modern and powerful GPUs that contain thousands of cores, such as the NVIDIA GeForce RTX 30 Series [3].

Compute Unified Device Architecture (CUDA) is a general-purpose parallel computing platform and programming model introduced by NVIDIA. CUDA made developing parallel GPU applications much easier [21]. A typical CUDA program executes on both the GPU (Device) and the CPU (Host). The code executed on the GPU is grouped into a special function determined by the "global" qualifier and launched by the kernel invocation. The kernel parameters determine the grid and block dimensions. Threads within the block are partitioned into warps. A warp is made up of 32 parallel threads that are all executed based on the single instruction multiple thread (SIMT) paradigm.

The CUDA kernel should be invoked by the host code to carry out parallel computations [22]. The kernel invocation is asynchronous with respect to the host. After the kernel call, the host code must use the "cudaDeviceSynchronize()" API function to make the invocation synchronous. This indicates that the host will wait until all GPU threads have finished running. This API function ensures implicit global synchronization within the grid [1].

When the computational algorithm requires some synchronization points, the developer should break it up into several steps. Each step is mapped to one device kernel. The invocation of each kernel should be followed by the cudaDeviceSynchronize() function. This mechanism will perform synchronization between the algorithm steps. The CUDA API also contains the "__syncthreads()" function to perform inter-block local synchronization among threads in the same block.

In CUDA 9, NVIDIA introduced the cooperative groups extension. This extension allows the programmer to synchronize all threads in the same group [2, 23]. The group could be the grid. In this way, the programmer would be able to synchronize all threads in all blocks.

One of the requirements to use cooperative groups is to launch the cuda kernel through the "cudaLaunchCooperativeKernel" API function. The drawback of cooperative group synchronization is the limited number of launched blocks per multiprocessor. It cannot exceed the maximum number returned by the "cudaOccupancyMaxActiveBlocksPerMultiprocessor" function. This function can provide an occupancy prediction based on the block size and shared memory usage of a kernel [21].

GPUs, with a compute capability of 3.0 or higher, provided a mechanism to allow threads to directly read another thread's register in the same warp. The shuffle instruction enables threads in a warp to interact with one another without using shared or global memory. It offers applications a quick way to exchange data among threads in a warp [1].

3.3. GENETIC ALGORITHM OVER GPU

It is not easy to implement a GA on GPGPU, and numerous implementations and models have been suggested and explored in earlier literature. When you apply

the genetic algorithm in a parallel way, more than one thread will participate in completing the steps of this algorithm. Attention should be paid to performing each step completely before moving on to the next step. This means inserting synchronization points between the steps [24]. There are three models of the parallel genetic algorithm (PGA) on GPGPU: the master-slave model, the island model, and the cellular model [7–9].

In the traditional master-slave model, there is a single population. The evaluation step is achieved on the GPU by using a single kernel. The other GA steps are achieved sequentially on the CPU [8]. This model requires one kernel call for each iteration as well as two operations to copy data from host to device and vice versa. In the master-slave model, there is the possibility of applying all GA steps in parallel on the CPU by employing multiple CPU threads [25]. It can also be done on a GPU by mapping each step to a separate kernel.

This means that each iteration involves multi-kernel invocations. This model is good for performing implicit global synchronization by using the cudaDeviceSynchronize() function. However, it is inefficient since it needs frequent CPU-GPU communication to launch kernels at each iteration [7]. The execution time is negatively affected by this communication.

In the traditional island model, the population is divided into subpopulations called islands [8]. Each subpopulation is kept in the shared memory and evolves separately in one block. This model includes an additional step called migration, which is carried out between neighboring islands every predetermined number of iterations.

This model doesn't require global synchronization following each genetic step because each island evolves independently. Inter-block synchronization is implemented after each step [17]. The migration step is an exception and involves global synchronization between all GPU blocks to migrate some individuals from an island to a neighboring island through the global memory. Therefore, each iteration that doesn't perform migration can be achieved with only a single kernel call.

4. PROPOSED APPROACHES

This section describes the procedures required to implement the lightweight island model (LIM). Mixing between the island model and persistent threads is implemented in two approaches. The first is the lightweight island model based on cooperative groups (LIM-CG), and the second is the lightweight island model based on implicit synchronization (LIM-IS). The scheme in both approaches will be discussed, in addition to the work distribution mechanism at the steps of the genetic algorithm. Since both approaches are based on the island model, the work distribution mechanism will be identical in both.

4.1. APPROACH SCHEMES

In the first approach (LIM-CG), grid synchronization is applied using cooperative groups. All steps of the genetic algorithm, including migration, are mapped to only one kernel (GAKernel1), as shown in Fig. 2.

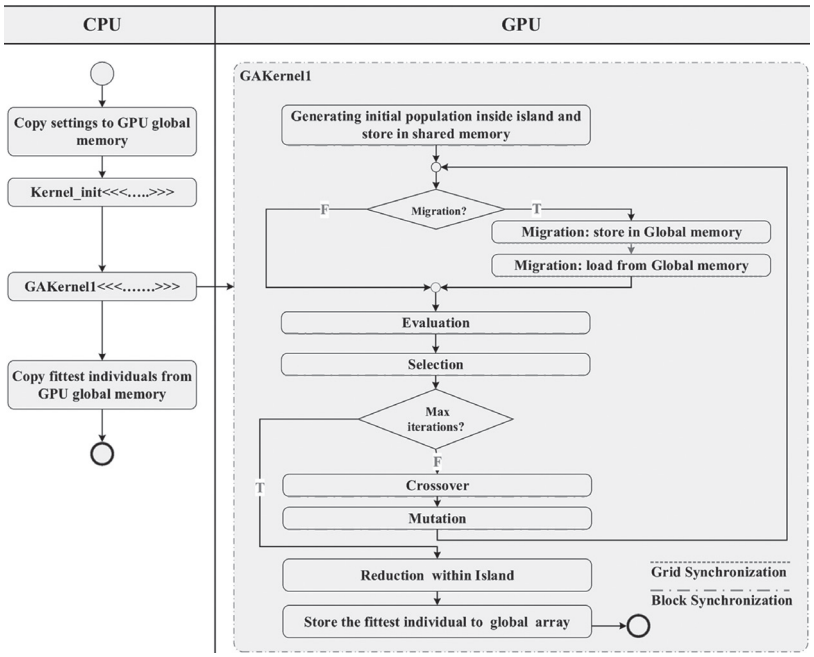

Fig. 2. Scheme of the lightweight island model based on cooperative groups (LIM-CG).

This kernel stays alive until it reaches the maximum number of iterations. This kernel code starts by generating the initial population. Each thread, in each block, is responsible for producing one individual through shuffle operations. The created individual is then stored in the island subpopulation array located in the shared memory. The "extern" identifier can be used to allocate this array. After that, all threads in the block enter the

evolving process (while loop). Migration is the first step that is encountered. It is performed frequently after a predetermined number of iterations (migration frequency). A global array is allocated in global memory to help migrate individuals between islands.

The second approach (LIM-IS) depends on performing an implicit global synchronization by returning control to the host after a predefined number of iterations. This number depends on the migration frequency. In this way, the number of global synchronization points is reduced. The migration kernel is invoked only when there is a migration. There are three kernels (InitialPopKernel, GAKernel2, and MigrationKernel3), as shown in Fig. 3.

The first one is responsible for generating the initial population. Each thread, in every block, is responsible for producing one individual, calculating the fitness, and then storing it in the global array. The second kernel repeats the steps of the genetic algorithms (while loop) until there is a need for migration (migration frequency). At that point, the control is returned to the host to launch the migration kernel just once. The loop counter at the host side is incremented by the value of migration frequency.

Synchronization inside every island is ensured by inter-block local synchronization. Global synchronization is implicitly guaranteed by returning the control to the host.

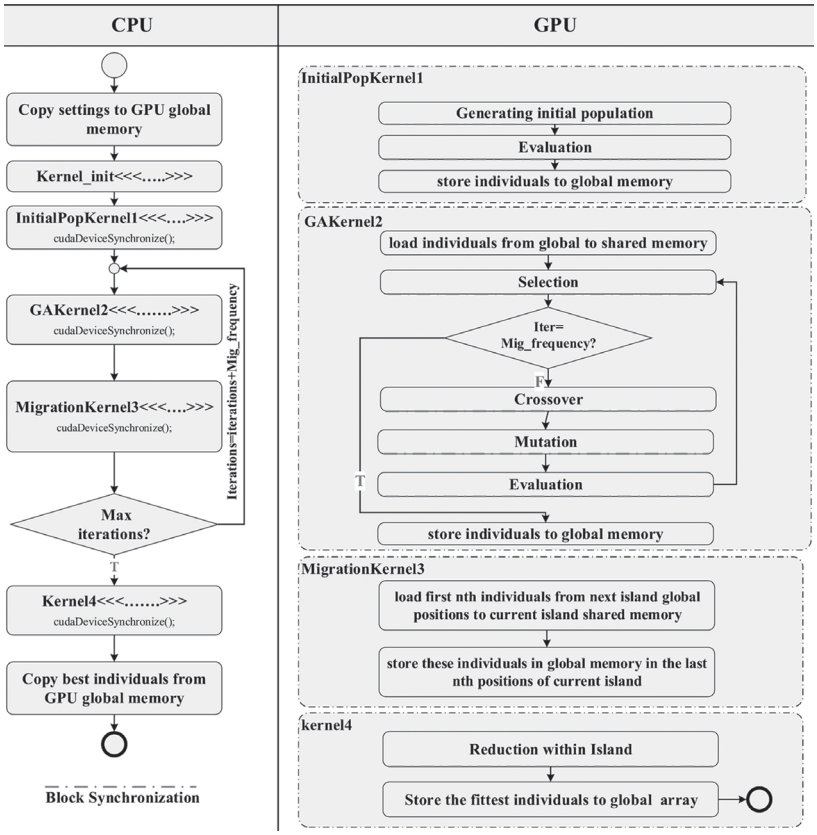

Fig. 3. Scheme of the lightweight island model based on implicit synchronization (LIM-IS).

Table 1. The number of synchronization points and the number of kernel invocations required in LIM-IS, LIM-CG, SAK, and SLK approaches.

Approach	Global synchronization points	Block synchronization points	Kernel invocations
SAK	#iterations * #steps		#iterations * #steps
SLK	#iterations * #steps		1
TIM	#iterations	#iterations * #steps	#iterations
LIM-CG	#iterations / #migrations	#iterations * #steps	1
LSM-IS	#iterations / #migrations	#iterations * #steps	#iterations / #migrations + #migrations

Migrating individuals between islands (blocks) is achieved through global memory. This approach does not rely on cooperative groups therefore, the number of blocks that can be executed on the device is unrestricted. It can be controlled either directly from the host or by a nested kernel. The host will be responsible for calling a single parent kernel, with one block and one thread, which in turn will invoke the other kernels.

Table 1 summarizes the synchronization points and the number of kernel invocations required. It compares the proposed approaches (LIM-IS, LIM-CG) with the switching among kernels approach (SAK) presented

in [13], the scheduled light kernel approach (SLK) presented in [10], and the traditional island model (TIM) presented in [14]. This is done without regard to the kernel that initializes the random vector.

In the SLK approach, they launched a single kernel with multiple blocks. All the blocks must be synchronized before moving toward the next scheduled computation (the next step). This style will increase the global synchronization points inside the kernel. In the SAK approach, there are three kernel invocations per iteration. This means three global synchronization points that are achieved by implicit synchronization.

In our approach, there is no need for global synchronization after each step. Inter-block synchronization, within the island, is performed after each step. It is guaranteed by the _syncthreads() function. Global synchronization is involved only when there is a migration between islands. Global synchronization is achieved through cooperative groups or implicit synchronization.

Several kernel invocations occur in the SAK model, and the number of threads participating in each step can differ. Although this paradigm reduces the number of kernel-running blocks, it increases the CPU-GPU communication overhead.

4.2. WORK DISTRIBUTION

When launching the persistent kernel in the SLK model, the maximum number of threads required for the overall steps is employed. A work scheduling matrix is used to distribute work among threads. It has several rows corresponding to the algorithm steps and several columns that define the working state of the block at this step. Some blocks can be assigned a no-operation (NOP) if they are not needed during a specific computation step.

In our proposed approaches, we'll try to effectively distribute the work among the threads to ensure that the most active threads are participating in the current step.

Work distribution is guaranteed by some variables. The value of these variables is determined by the probability of the genetic operators, the length of the route, and the configuration of the islands.

As illustrated in Pseudo-code 1, these variables are grouped into a structure called settings that is initially copied to the device. At each step, some equations are calculated using these variables to define the number of threads participating and how the work will be distributed among them.

Crossover and selection operations require the most computation time in comparison to the other steps. The one-point crossover is used in the crossover operator, while the tournament selection method is used in the selection step since it's preferable for parallel implementation.

In the selection step, the island population is divided into several groups depending on the number of selected individuals, needed and the number of warps inside the island.

Pseudo-code 1. The structure of the settings.

```
struct settings
{
    float CROSSOVER_RATE;
    float MUTATION_RATE;
    float SELECTION_RATE;
    float MIGRATION_RATE;
    int MAX_ITERATIONS;
    int ISLAND_SIZE;
    int ROUTE_SIZE;
    int NUM_ISLANDS;
    int MIGRATION_FREQ;
};
```

As seen in Fig. 4, each warp will manipulate one group to find the fittest individuals inside it through an unrolled reduction operation.

The reduction stops when it reaches the fittest individuals involved. Then, based on the warp and thread indices, the fittest individuals will be stored in the proper positions in the island population.

To facilitate the task, an alternative matrix is used, in which we store the fitness values of the individuals and their indices. Pseudo-code 2 describes the selection step.

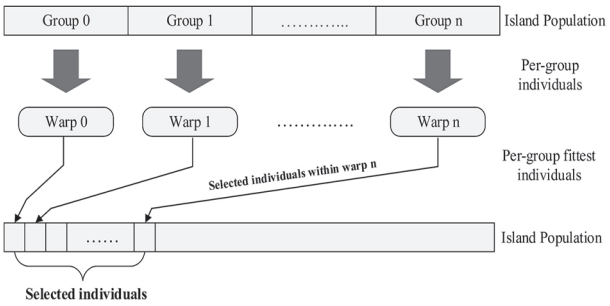

Fig. 4. Work distribution inside the island in the selection step.

The one-point crossover process takes place in two phases. In the first phase, the points located before the crossing point are transferred from the first parent to the child. In the second phase, all the points of the second parent are tested to see if they are not duplicated in the child. Then the non-duplicated points are added to the child after the crossing point.

The second stage requires many comparisons to be completed, depending on the length of the route. For this reason, we will allocate a warp to accomplish the crossover process between two parents to generate a child, as shown in Fig. 5.

At the start, Thread0 within the warp generates the index of the crossover point, the position for parent1, and the position for parent2.

Pseudo-code 2. The selection step.

```
#SelectedIndiveduals = SELECTION_RATE * ISLAND_SIZE #warpsInIslad= ISLAND_SIZE /32
WarpIndex= trunc(thid / 32)
#SelectionOperationsInsideWarp=ceil  (#SelectedIndiveduals / #warpsInIsland)
Index= WarpIndex * #SelectionOperationsInsideWarp
Unrolled Reduction Inside the group
Store fittest individuals to shared memory starting from index
```

Pseudo-code 3. The crossover step.

```
#CrossedIndiveduals= CROSSOVER_RATE * ISLAND_SIZE
#warpsInIslad= ISLAND_SIZE /32
WarpIndex= trunc(thid / 32)
#CrossoverOperationsInsideWarp=ceil (#CrossedIndiveduals / #warpsInIsland)
ChildIndex= #SelectedIndiveduals + WarpIndex * #CrossoverOperationsInsideWarp
for(j=0;j<#CrossoverOperationsInsideWarp;j++)
    CrossPosition=generatRandom (1,ROUT_SIZE-1)
    parent1Index= generatRandom (0,#selectedIndividuals)
    parent2Index= generatRandom (0,#selectedIndividuals)
    CrossPosition = __shfl_sync (0xFFFFFFFF,CrossPosition,0,32)
    parent1Index = __shfl_sync (0xFFFFFFFF,parent1Index, 0, 32)
    parent2Index = __shfl_sync (0xFFFFFFFF,parent2Index, 0, 32)
    read parent1, parent2
    stride=32
    threadIndex=threadIdx.x%32
    for(i= threadIndex; i< CrossPosition ; i=i+stride)
       if(i< CrossPosition)
              IslandPop[ChildIndex].rout[i]=parent1.rout[i]
    GroupSize = ceil(ROUT_SIZE/32)
    for(i= threadIndex * GroupSize;i<threadIndex * GroupSize+ GroupSize;i++)
       Test duplicate points in parent2[i] with parent1[1, CrossPosition]
    Store non-duplicated points to shared memory
    Thread0 will update the child located at the ChildIndex position
    childIndex= childIndex+1
```

Then it broadcasts these values to all of the threads in the same warp via the shuffle operation.

The threads within the warp will participate in transferring the points from the first parent to the child. This will decrease the divergence among threads in the same warp. After that, the points of the second parent will be divided and distributed among the threads to be tested for doubling in the child.

The value of the non-duplicated point, or -1, will be recorded to indicate the presence of doubling.

This process is done by using the data of the first father [0, crossing position] that is stored with each thread to avoid saving multiple copies of the child.

Finally, the non-duplicated points will be stored in shared memory. Thread0 inside the warp will update the points of the child in the population after the crossing position. Pseudo-code 3 describes the details of the crossover step.

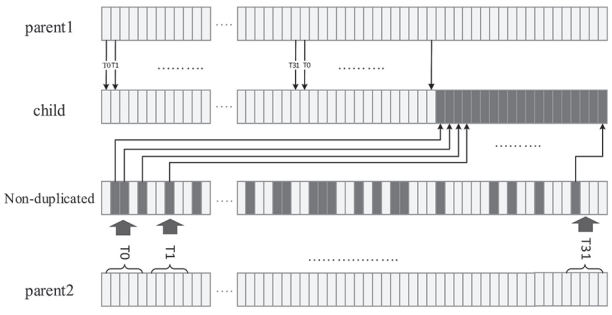

Fig. 5. Work distribution in the crossover process

5. EXPERIMENTAL RESULTS

This section presents the results of the two presented approaches (LIM-IS, LIM-CG) compared with the serial CPU, the switching among kernels approach (SAK) presented in [13], the scheduled light kernel approach (SLK) presented in [10], and the traditional island model (TIM) presented in [14].

The experiments were implemented on a laptop with an Intel Corei5-10 2.5GHz CPU and Nvidia RTX 3050Ti GPU. This GPU has 20 SMs, which means a total of 2560 CUDA cores. The compilation was performed using Microsoft Visual Studio 2019 with CUDA 11.6 SDK. The experiments are carried out with various numbers of population-size.

Since we applied warp-based parallelism, as explained in Section 4, the block-size may vary from the island-size. Where a single individual is mapped to several threads. In order to determine the GPU configurations (grid-size, block-size) that lead to the best possible performance, the first experiment will test the performance of our island-based approaches (LIM-IS, LIM-CG) against the serial CPU implementation. The number of individuals per island (island-size) is set to 128. Each block manipulates a single island. The grid-size determines the number of running blocks (islands). The population-size results from island-size*128. The same settings, listed in Table 2, were adopted during the implementation.

Table 2. Parameters of the TSP and the genetic algorithm

Parameter	Value
Problem	Att48
Maximum iterations	1000
Selection	Tournament selection (30%)
Crossover	One-point crossover (65%)
Mutation	Swap mutation (5%)
Migration topology	Unidirectional ring
Migration frequency	Every 50 iterations

The results of the first experiment are displayed in Table 3. Results show that choosing a block-size=1024 will provide the best performance compared to the serial implementation. Taking into account that the maximum number of threads that can run on the GPU being used is 1024 threads per block.

This experiment also proves that the GPU implementation gives a significant improvement in execution time compared to the serial CPU implementation. Acceleration of up to 27x has been achieved using the GPU, knowing that the population-size and algorithm parameters are the same in both cases.

In the second experiment, we will compare our approaches to those of SAK, SLK, and TIM for different population sizes. The GPU configuration (grid-size, block-size), as described in Section 2, was different for each approach. This relates to the population-size that was being used in the experiment.

To demonstrate the experimental process, Table 4 displays the necessary GPU settings to test each of the aforementioned approaches on the same population size. Each thread in the TIM, SLK, and SAK approaches, was mapped to a single individual. The number of GPU running threads must be equal to the population-size. In our approaches, and based on the results of the first experiment, the block-size is set to 1024.

The experiments were carried out with various numbers of population-size. The GA-TSP parameters, listed in Table 2, were adopted during the implementation of serial CPU and parallel GPU approaches.

Table 3. The execution time and speedup of the LIM-CG and LIM-IS approaches for different numbers of block-size and population-size

Block-size	Grid-size	Island-size	Population-size	Serial time (ms)	LIM-CG		LIM-IS	
					Time (ms)	Speedup	Time (ms)	Speedup
128	8	128	1024	1434.62	1304.20	1.1	2049.45	0.7
	16	128	2048	2857.78	2198.29	1.3	2597.98	1.1
	32	128	4096	5661.31	2358.88	2.4	2461.44	2.3
	64	128	8192	11278.44	3759.48	3	3638.21	3.1
	128	128	16384	22685.26	5532.99	4.1	4931.58	4.6
	256	128	32768	45543.33			8433.95	5.4
256	8	128	1024	1434.62	1024.73	1.4	1103.55	1.3
	16	128	2048	2857.78	893.06	3.2	952.59	3
	32	128	4096	5661.31	1286.66	4.4	1347.93	4.2
	64	128	8192	11278.44	1819.10	6.2	2128.01	5.3
	128	128	16384	22685.26	3287.72	6.9	3065.58	7.4
	256	128	32768	45543.33			5117.23	8.9
512	8	128	1024	1434.62	843.89	1.7	896.63	1.6
	16	128	2048	2857.78	476.30	6	529.22	5.4
	32	128	4096	5661.31	602.27	9.4	622.12	9.1
	64	128	8192	11278.44	989.34	11.4	947.77	11.9
	128	128	16384	22685.26	1731.70	13.1	1786.24	12.7
	256	128	32768	45543.33			2828.78	16.1
1024	8	128	1024	1434.62	531.34	2.7	683.15	2.1
	16	128	2048	2857.78	280.17	10.2	357.22	8
	32	128	4096	5661.31	339.00	16.7	365.25	15.5
	64	128	8192	11278.44	433.79	26	458.47	24.6
	128	128	16384	22685.26	807.30	28.1	840.19	27
	256	128	32768	45543.33			1615.01	28.2

Table 5 and Fig. 6 display the execution times for each approach. Fig. 7 compares the speedup of the suggested approaches with serial execution and earlier-mentioned studies.

The results shown in Fig. 6 and Fig. 7 demonstrate that the two new approaches can increase the speed-up to 4.5x over the TIM, and up to 1.5x– 2x over SLK and SAK approaches.

Table 4. The GPU configuration.

Approach	GPU configurations
LIM-CG LIM-IS	Island-size: 128 Grid-size: Population-size/128, Block-size: 1024
TIM	Island-size: 128 Grid-size: Population-size/128, Block-size: 128
SLK	Grid-size: 40 (twice the number of available SMs) Block-size: population-size / grid-size.
SAK	Grid-size: population-size / block-size. Block-size: 1024 (The maximum number allowed)

Table 5. The execution time (ms) of the proposed approaches LIM-IS and LIM-CG compared with SAK, SLK, and TIM approaches.

Population-size	Serial	TIM	SAK	SLK	LIM-CG	LIM-IS
1024	1434.6	1434.6	1024.7	478.2	531.3	683.1
2048	2857.7	1512.9	697.1	453.6	280.1	357.2
4096	5661.3	2358.8	775.5	496.6	339.0	365.2
8192	11278.4	2819.6	848.0	751.9	433.7	458.4
16384	22685.2	3979.8	1597.5	1281.6	807.3	840.1
32768	45543.3	7116.1	3098.1	2384.4		1615.0

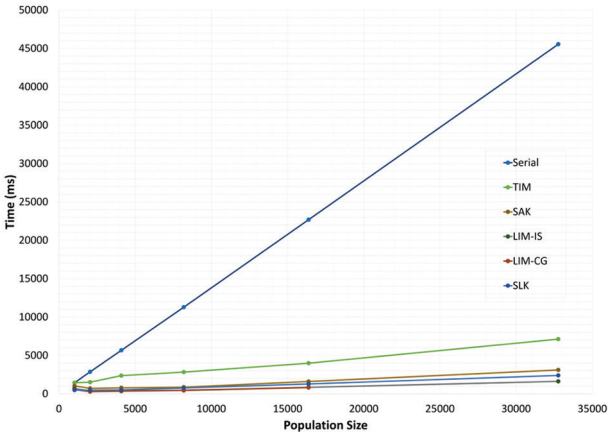

Fig. 6. The execution time of the proposed approaches LIM-IS and LIM-CG compared with SAK, SLK, and TIM approaches.

It can be seen that the first approach, which uses co-operative groups, is limited to a maximum number of blocks within the device.

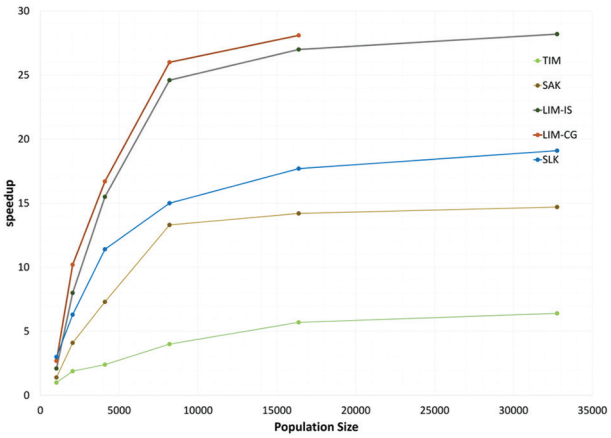

Fig. 7. The speedup achieved by the proposed approaches LIM-IS, and LIM-CG compared with SAK, SLK, and the TIM approaches

The last experiment will focus on evaluating the quality of the obtained solutions. The parameters and settings applied in this experiment are those listed in Table 2 and Table 4. The relative error between the fittest solution's cost and the optimal solution's cost, represented by equation (1), was calculated. The execution was repeated ten times, and the average value was recorded.

$$Relative\ error = \left| \frac{Cost_{Fittest} - Cost_{Optimal}}{Cost_{Optimal}} \right| \quad (1)$$

Fig. 8 displays the relative error resulting from serial CPU and GPU-based parallel implementations. It has been observed that the quality of the solution improves when the size of the population increases. The island-based approaches (TIM, LIM-CG, LIM-IS) have an advantage in obtaining the highest quality because of the migration step. Migrating some individuals between islands gives them the possibility to explore different regions of the search space and discover better-quality solutions.

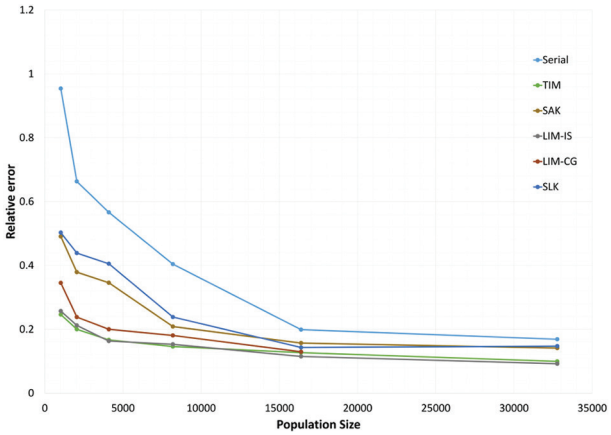

Fig. 8. The relative error of the solutions that resulted from serial, LIM-IS, LIM-CG, SAK, SLK, and TIM approaches.

Noting that the experiments were carried out for 1000 iterations for all aforementioned approaches.

6. CONCLUSION AND FUTURE WORK

In this paper, we highlight the details of implementing a lightweight island model (LIM) on a general-purpose graphic processor unit (GPGPU) for the TSP problem. Two approaches were suggested (LIM-IS, LIM-CG) that follow the concept of persistent threads. The necessary details to convert the traditional island model (TIM) into a lightweight island version were discussed. We also reviewed the previously suggested researches that worked on transforming the traditional master-slave model into new approaches, such as the switching among kernels approach (SAK) and the scheduled light kernel approach (SLK). A new mechanism was presented for distributing work between live threads inside the islands. The GPU configurations were tested and detailed to get the best possible performance. The introduced approaches were compared, in terms of execution time and solution quality, with serial implementation and with previous works including the traditional island model (TIM), switching among kernels approach (SAK), and scheduled light kernel approach (SLK).

Our suggested approaches produced much better results compared with these previous works. The speedup achieved is up to 27x compared with the serial CPU and up to 4.5x compared to the TIM approach. The speedup improvement was up to 1.5– 2x over SLK and SAK approaches.

In terms of solution quality, the proposed approaches produced better solutions than SAK and SLK, whereas the results were nearly identical between the proposed approaches and TIM.

For future work, these approaches can be tested on larger TSP problems to measure the effectiveness of the synchronization methods and the parallelization strategy.

7. REFERENCES

[1] J. Cheng, M. Grossman, T. McKercher, "Professional CUDA C Programming", Wiley, 2014.

[2] G. Barlas, "Multicore and GPU Programming: An Integrated Approach", 2nd Edition, Elsevier, 2022.

[3] NVIDIA, "GeForce RTX 30-Series Laptops | NVIDIA", https://www.nvidia.com/en-me/geforce/gaming-laptops/. (accessed: 2022)

[4] B. Basbous, "2D UAV Path Planning with Radar Threatening Areas using Simulated Annealing Algorithm for Event Detection", Proceedings of the International Conference on Artificial Intelligence and Data Processing, Malatya, Turkey, 28-30 September 2018, pp. 1-7.

[5] M. Cakir, "2D path planning of UAVs with genetic algorithm in a constrained environment", Proceedings of the 6th International Conference on Modeling, Simulation, and Applied Optimization, Istanbul, Turkey, 27-29 May 2015, pp. 1-5.

[6] P. Larrañaga, C. M. H. Kuijpers, R. H. Murga, I. Inza, S. Dizdarevic, "Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators", Artificial Intelligence Review, Vol. 13, No. 2, 1999, pp. 129-170.

[7] J. R. Cheng, M. Gen, "Accelerating genetic algorithms with GPU computing: A selective overview", Computers & Industrial Engineering, Vol. 128, 2019, pp. 514-525.

[8] J. R. Cheng, M. Gen, "Parallel Genetic Algorithms with GPU Computing", Industry 4.0 - Impact on Intelligent Logistics and Manufacturing, Vol. 32, 2020, pp. 137-144.

[9] T. Harada, E. Alba, "Parallel Genetic Algorithms: A Useful Survey", ACM Computing Surveys, Vol. 53, No. 4, 2021, pp. 1-39.

[10] N. Capodieci, P. Burgio, "Efficient Implementation of Genetic Algorithms on GP-GPU with Scheduled Persistent CUDA Threads", Proceedings of the Seventh International Symposium on Parallel Architectures, Algorithms and Programming, Nanjing, China, 12-14 December 2015, pp. 6-12.

[11] R. Saxena, M. Jain, S. Bhadri, S. Khemka, "Parallelizing GA based heuristic approach for TSP over CUDA and OPENMP", Proceedings of the International Conference on Advances in Computing, Communications and Informatics, Udupi, India, 13-16 September 2017, pp. 1934-1940.

[12] M. Oiso, Y. Matsumura, T. Yasuda, K. Ohkura, "Implementation Method of Genetic Algorithms to the CUDA Environment using Data Parallelization", Tehnički Vjesnik, Vol. 18, No. 4, 2011, pp. 511-517.

[13] M. Abbasi et al. "An efficient parallel genetic algorithm solution for vehicle routing problem in cloud implementation of the intelligent transportation systems", Journal of Cloud Computing, Vol. 9, No. 1, 2020.

[14] T. Van Luong, N. Melab, E.-G. Talbi, "GPU-based island model for evolutionary algorithms", Proceedings of the 12th annual conference on Genetic and evolutionary computation, 2010, pp. 1089-1096.

[15] X. Sun, L.-F. Lai, P. Chou, L.-R. Chen, and C.-C. Wu, "On GPU Implementation of the Island Model Genetic Algorithm for Solving the Unequal Area Facility Layout Problem", Applied Sciences, Vol. 8, No. 9, 2018, p. 1604.

[16] B. Wang et al. "Multipopulation Genetic Algorithm Based on GPU for Solving TSP Problem", Mathematical Problems in Engineering, Vol. 2020, 2020.

[17] D. M. Janssen, A. W. C. Liew, "Acceleration of genetic algorithm on GPU CUDA platform", Proceedings of the 20th International Conference on Parallel and Distributed Computing, Applications and Technologies, Gold Coast, QLD, Australia, 5-7 December 2019, pp. 208-213.

[18] P. Pospichal, J. Schwarz, J. Jaros, "Parallel genetic algorithm solving 0/1 knapsack problem running on the GPU", Mendel, 2010, pp. 64-70.

[19] K. Gupta, J. A. Stuart, J. D. Owens, "A study of Persistent Threads style GPU programming for GPGPU workloads", Proceedings of Innovative Parallell Computing, San Jose, CA, USA, 13-14 May 2012, pp. 1-14.

[20] D. E. Goldberg, J. H. Holland, "Genetic Algorithms in Search, Optimization, and Machine Learning", Machine Learning, Vol. 3, No. 2, 1979, pp. 95-99.

[21] NVIDIA, "Cuda C Programming Guide", 2017, pp. 1-261.

[22] T. Soyata, "GPU Parallel Program Development Using CUDA", CRC Press, Chapman and Hall/CRC, 2018.

[23] NVIDIA, "Programming Guide :: CUDA Toolkit Documentation", https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cooperative-groups (accessed: 2022)

[24] S. Chen, S. Davis, H. Jiang, A. Novobilski, "CUDA-Based Genetic Algorithm on Traveling Salesman Problem", Studies in Computational Intelligence, Vol. 364, 2011, pp. 241-252.

[25] M. AlRaslan, A. H. AlKurdi, "UAV Path Planning using Genetic Algorithm with Parallel Implementation", International Journal of Computer Science and Information Technologies, Vol. 10, No. 02, 2022, pp. 1-15.