Acceleration of a CUDA-Based Hybrid Genetic Algorithm and its Application to a Flexible Flow Shop Scheduling Problem

Didier El Baz, Jia Luo, Jinglu Hu

Acceleration of a CUDA-Based Hybrid Genetic Algorithm and Its Application to a Flexible Flow Shop Scheduling Problem

Jia Luo, Didier El Baz
LAAS-CNRS, Universite de Toulouse, CNRS
Toulouse, France
jluo@laas.fr, elbaz@laas.fr

Jinglu Hu
Graduate School of Information, Production and Systems,
Waseda University
Fukuoka, Japan
jinglu@waseda.jp

***Abstract*—Genetic Algorithms are commonly used to generate high-quality solutions to combinational optimization problems. However, the execution time can become a limiting factor for large and complex problems. In this paper, we propose a parallel Genetic Algorithm consisting of an island model at the upper level and a fine-grained model at the lower level. It is designed to be highly consistent with the CUDA framework to get the maximum speedup without compromising to solutions' quality. As several parameters control the performance of the hybrid method, we test them by a flexible flow shop scheduling problem and analyze their influence. Finally, numerical experiments show that our approach cannot only obtain competitive results but also reduces execution time by setting a medium size selection diameter, a relatively large island size and a wide range size migration interval.**

Keywords—Parallel Genetic Algorithm; GPU computing; CUDA; Flexible Flow Shop Scheduling

I. INTRODUCTION

The Genetic Algorithm (GA) is a stochastic search algorithm based on the principle of natural selection and recombination [1]. Nevertheless, there is an increase in the required time to find adequate solutions when GAs are applied to complex and large problems. Parallel implementation is considered as one of the most promising options to make it faster. There are different ways of exploiting parallelism in GAs [2]: master-slave models, fine-grained models, island models, and hybrid models. The master-slave model is the only one that does not affect the behavior of the algorithm by distributing the evaluation of fitness function to slaves. The fine-grained model works with a large spatially population. The evolution operations are restricted to a small neighborhood with some interactions by overlap structure. The island model divides population into subpopulations. These subpopulations on independent islands are free to converge towards different sub-optima and a migration operator can help mix good features that emerge from the local islands. The hybrid model combines any two of the above methods.

Regarding to solve scheduling problems by parallel GAs, various researches have been done with different architectures [3][4][5]. We can discover that some works pay heavier attention on speedup gained from the parallelization. On the opposite, the others consider more the improvement for solutions' quality and convergence speed. Few implementations have discussed them simultaneously with a fair comparison.

In the last decade, Graphics Processing Units (GPU) have gained widespread popularity as computing accelerators. To achieve general-purpose parallel computation on GPU, the Compute Unified Device Architecture (CUDA) [6] was developed in 2006. It is a framework that takes the maximum advantage of the low-lying hardware using an industry standard programming language [7]. Some researches have tried to parallelize GAs on GPU with CUDA [8]. However, the implementation to solve scheduling problems is rare. Besides, the aim of such an approach is usually to get the maximum possible speedup while compromising the solutions' quality.

Therefore, designing a parallel GA that is highly consistent with the CUDA framework balancing conflicts between the solutions' quality and the execution time remains an open research challenge. In this paper, we seek to address it and its application to a flexible flow shop scheduling problem (FFS). Specially, the contributions of our work are summarized as follows:

- We design a hybrid GA consisting of an island model at the upper level and a fine-grained model at the lower level respecting to the CUDA framework.
- As three parameters control the performance of the proposed method, a configuration test has been carried to fully understand their influence.
- Comparative experiments witness that our approach can not only obtain competitive results but also reduces execution time.

The remaining sections of this paper are organized as follows. Section 2 introduces related works. Section 3 describes the FFS definition. Section 4 presents the design of the parallel GA and its adaptable operators. Section 5 illustrates the numerical experiments and result analysis. Finally, conclusions are stated in section 6.

II. RELATED WORKS

There have been extensive research in parallel GAs to solve shop scheduling problems. A master-slave GA for a flow shop

problem was presented in [9]. The cooperation between a single population and a group of local subpopulations was implemented on a laptop with Prentium IV core 2 Duo 2.53 GHz CPU. A fine-grained GA solving job shop scheduling problems was considered by Tamaki et al. [10]. The method was modified as an absolute neighborhood model and implemented on Transputer, where the selection was executed locally in a neighborhood of each population. Harmanani et al [11] discussed an island GA to solve a non-preemptive open shop scheduling problem. In this paper, islands were connected through an Ethernet network and the Message Passing Interface was used for migration on a cluster of five machines. Although the master-slave model carries out the exact same search as the classical GA, frequent data transfers between the host and the device generate a bottleneck. Other methods change how the GA works. Despite the fact that the island model dominates the work on parallel GAs, it is hard to conclude that the island model overcomes other models since the comparison cannot be made in absolute terms.

Research on CUDA-based GAs has won favor in recent years. Pospichal et al. [12] presented an implementation of a parallel island-based GA with unidirectional ring migrations on CUDA. Munawar et al. [7] designed a hybrid of fine-grained GA and local search to solve a maximum satisfiability problem using CUDA. Zhang et al. [13] proposed a hierarchical GA implemented by CUDA mixing an island model and a master-slave model. However, a few works have been carried to solve shop scheduling problems by CUDA-based parallel GAs. AitZai et al. [14] studied a job shop scheduling problem with blocking by a mater-slave GA with some memory management respecting to the CUDA framework. Numerical tests displayed the proposed method using GPU got maximum 15 times more explored solutions than the GA using CPU in a limited execution time. Zajicek et al. [15] proposed a homogeneous parallel GA model on the CUDA architecture. The main idea was based on an island GA and some instances of the flow shop scheduling problem were solved with speedup from 60 to 120 comparing to the equivalent sequential CPU version. Although CUDA is working with the two-dimensional grid environment that matches well the design of the fine-grained GA, there is still no research implement the fine-grained GA with CUDA to solve shop scheduling problems with the best of our knowledge.

No matter the amount of related papers or the various types of treated problems, the implementation of CUDA-based GA for shop scheduling problems is rare. Besides, none of them so far has focused on the effects of multiple controlling parameters as far our knowledge is concerned. The above-mentioned efforts provide us a starting point to design a CUDA consistent GA for shop scheduling problems. This method is supposed to achieve a balance between the solutions' quality and the execution time by fully understanding the effects of its controlling parameters.

III. Problem Definition

A classical flow shop scheduling problem (FFS) with the objective to minimize the total tardiness and the makespan is represented by $WT * \sum T_j + \mathrm{C}_{\max}$ using the classification scheme of Bruzzone et al. [16], where WT indicates the priority of the first objective. The FFS is a multistage production process that involves two or more stages in series. There is at least one machine in each stage, and at least one stage has more than one machine. All jobs need to go through all stages in the same order before they are completed. On each stage, one machine is selected for processing a given operation from one job. An instance of the FFS problem considers a set of J jobs $(1 \le j \le J)$. Each of them consists of a set of S stages $(1 \le s \le S)$. At every stage, there is a set of M machines $(1 \le m \le M)$. The processing time of job j at stage s on machine m is denoted as an operation and is abbreviated by (j, s, m). Usually, it is given in advance as P_{jsm} with the release time R_j and the due time D_j. A feasible solution is described by jobs' sequence on target machines M_{js}. To simplify the problem, we set each stage has the same number of machines and a mathematical model is proposed in Appendix.

IV. CUDA-based Parallel Genetic Algorithm

A. Hybrid Model

The parallel threads of CUDA are grouped into blocks that are organized in a grid as shown in Fig. 1. The hierarchisation of threads is related to the memory hierarchy. There are three distinct levels of memory [17]. The global memory is accessible to all threads. It is the largest memory of CUDA, but exhibits the highest latency. The shared memory enables threads only within a block whose access is much faster than the global memory. The local memory presents the lowest latency whereas it is only available to one thread with few KB of storage.

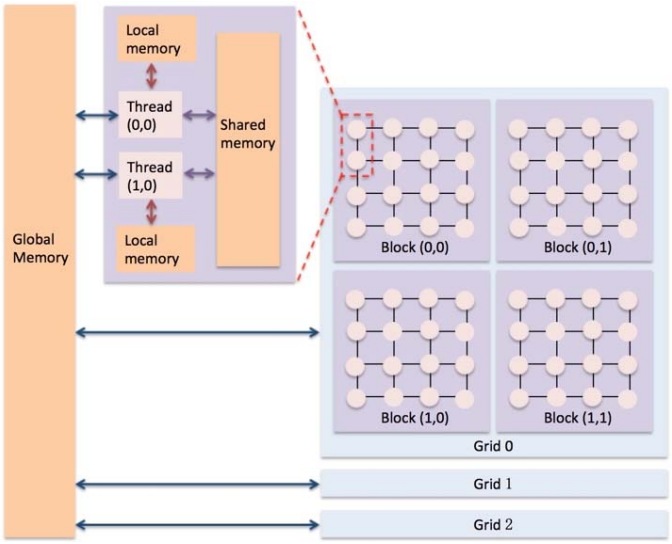

Fig. 1. Hierarchy of threads and different types of memory of CUDA

With respect to the CUDA framework, our parallel GA consists of two levels, a fine-grained GA at the lower level and an island GA at the upper level, as presented in Fig. 2. At the lower level, each CUDA thread processes one GA individual. Because of the 2D grid, the GA individuals can get connected completely. Selection, crossover, mutation and fitness value calculation are generated mainly via the local memory to enjoy its lowest latency unless imperative information exchange among individuals is done through the global memory. On the

other hand, one block on CUDA handles one island from GA at the upper level. An elitist strategy is carried out within the island using the shared memory after GA operators. Islands are interconnected with a single ring. An island can accept an individual with the best fitness value from the neighbor to overwrite the one with the worst fitness value as migration. The shared memory is chosen to complete this work primarily while the overwriting is processed via the global memory synchronously.

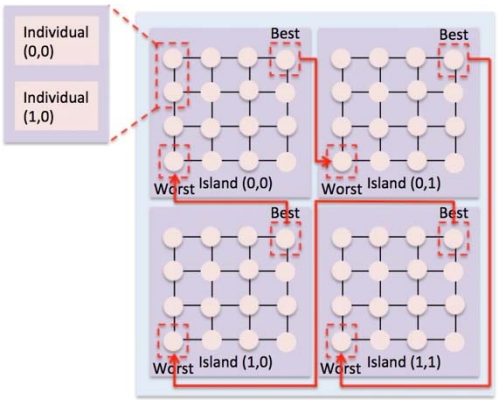

Fig. 2. Hierarchy of the hybrid GA

According to the evolution theory and the underlying architecture of CUDA, several advantages can be gained by the hybrid model over the classical one:

- At the lower level, the fine-grained model obtains good population diversity by dealing with high-dimensional variable spaces [18]. Limitation of interactions among individuals prevents the premature convergence. A reasonable neighborhood size with GA operators may disseminate the good solutions across the entire population.
- At the upper level, the island model increases the converge speed by subpopulations as the long-held principle in Population Genetics: favorable traits spread faster when the demes are small than when the demes are large [2]. An appropriate island size with a proper migration interval is able to optimize this performance.
- CUDA is build up with two-dimensional grid environment that matches perfectly the structure of the fine-grained GA. Thousands of CUDA threads are powerful to deal with large size populations concurrently without increasing time complexity. Meanwhile, the GA operations are carried using the fastest local memory.
- As CUDA threads are grouped into blocks, they are compatible to the mechanism of the island GA that divides the population into a few relatively large subpopulations. Isolated islands work on blocks in parallel with the help of shared memory to exchange information by migration.

B. GA operators

As selection and crossover operations of the classical GA need global information, they are not suitable to be moved over SIMT architectures without modification. On the opposite, mutation and fitness value calculation perform independently and can be easily parallelized on CUDA. The full implementation of the parallel GA is described in Table 1 where kernels launched on GPU are reflected as <<< >>>.

1) Encoding representation and fitness function

The CUDA based GA starts with a randomly generated initial population consisting of a set of individuals. An individual is represented by a chromosome. For the FFS, a chromosome consists of a string of length J×S, and the i-th gene states the index of the target machine for job $\lfloor i/S \rfloor + 1$ at stage $\{i/S\}+1$. As a minimization problem, the fitness function FIT(i) of an individual i is transferred from the objective function as $\max(E_{max} - (WT * \sum T_j + C_{max}, 0)$, where E_{max} is the estimated maximum value of the objective function.

2) Selection

The traditional selection operation is modified to suit the neighborhood configuration as in Fig.3. The selection area is defined with a certain diameter where the target individual is placed at the center of a grid. Among individuals within the selection area, the tournament selection is used where the individual with the best fitness value is selected to replace the target one.

TABLE I. PSEUDO-CODE FOR THE CUDA-BASED HYBRID GA

```
1:  initialize ();
2:  while (termination criteria are not satisfied) do
3:    generation++
4:    <<<island number, individual number/island>>>(population, selection diameter)   //selection operator
5:    <<<island number, individual number/island>>>(population, crossover rate)   //crossover operator
6:    <<<island number, individual number/island>>>(population, mutation rate)   //mutation operator
7:    <<<island number, individual number/island>>>(population, objective function information)   //fitness value calculation
8:    <<<island number, individual number/island>>>(population, history best individual per island)   //elitist strategy
9:    if (generation % migration interval==0)
10:     <<<island number, individual number/island>>>(population)   //migration
11:   end if
12: end while
```

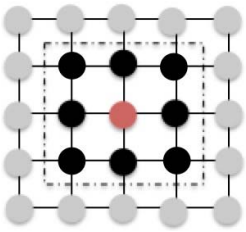

Fig. 3. Selection operation of the hybrid GA

3) Crossover

The individuals for crossover are paired with neighbors as in Fig.4 rather than mating two from the population randomly. Afterwards, a single point crossover is executed if a specified crossover probability is satisfied.

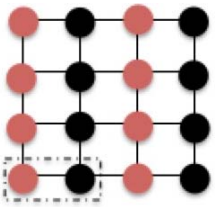

Fig. 4. Crossover operation of the hybrid GA

4) Mutation

Any individual in the population gets a random number generated on the interval 0 to 1. If it is smaller than the default mutation rate, genes in the chromosome are replaced by random values in the range, apart from the original ones.

V. Numerical Experiments

To analyze the performance of the proposed algorithm, all instances are characterized by 50 jobs with 4 stages and 3 available machines at each stage. The processing times P_{jsm} follow a uniform distribution U[1,5]. The jobs' release times R_j are generated from U[0, $\bar{P}$], where $\bar{P}=\sum_j \sum_s (\sum_m P_{jsm}/S)$. The jobs' due times are set as $D_j = R_j + \bar{P}_j(1+\sigma)$, where σ=U[0,2] and $\bar{P}_j = \sum_s (\sum_m P_{jsm}/S)$. Moreover, the weight WT for the total tardiness in the objective function is defined as 100. The crossover rate and the mutation rate are set as 0.9 and 0.1, respectively.

The experimental platform is based on the Intel Xeon E5640 CPU with 2.67GHz clock speed. The GPU code is carried out using CUDA 8.0 on NVIDIA Tesla K40, with 2880 cores at 0.745GHz and 12 GB GDDR5 global memory. All programs are written in C, except for the GPU kernels in CUDA C. All results display the average value of 100 runs.

A. Controlling Parameters Test

As the maximum threads amount per block on CUDA is 1024, we keep the population size as 1024 (32×32). There are three controlling parameters in this proposed method: the island size, the selection diameter and the migration interval. They are set by different numbers: island size (IS) = 4 (2×2), 16 (4×4), 64 (8×8), 256 (16×16), 1024 (32×32) individuals, selection diameter (SD) = 3, 9, 15, 21, 27 individuals, migration interval (MI) = 50, 40, 30, 20, 10 generations. Fig. 5 illustrates the convergence trend with combinations of different values. As the graph with all parameters' setting is a little confusing as shown by the first one. We separate it as 7 sub-graphs.

The island size keeps increasing from the second sub-graph to the sixth. It shows that the small selection diameter will lead to an early convergence, whereas there is not much improvement after it reaches a medium size. Meanwhile, this influence is more distinct when the island size is larger. As larger selection area requires larger memory, we suggest a medium size diameter value for implementations. For the following two tests, we keep its value as 9. The last two sub-graphs display the influence of the island size. As a result, a relatively larger island size makes the performance better. This trend is more obvious with groups of medium and large size selection diameters. Since the maximum threads amount per block on CUDA is 1024, the best performance is achieved by the island size 32×32. Moreover, there is no significant change when the migration interval is increased. Due to the additional cost caused by migration, it is advised not to have small migration intervals.

B. Comparison Test on Solution Quality

The classical GA works with a roulette wheel selection and a single point crossover pairing individuals randomly from the population, while the mutation keeps the same as the CUDA-based GA. We compare the average result and the best result between them as displayed in Fig. 6.

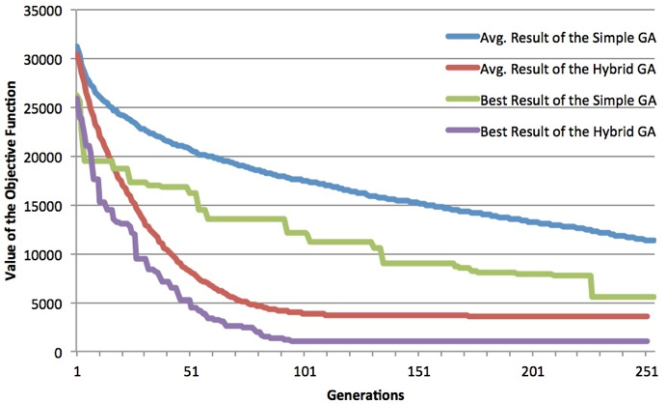

Fig. 6. Solution Quality Comparison

Fine-grained models obtain good population diversity when dealing with high-dimensional variable spaces [18] and island models converge faster by subpopulations [2]. By combining the merits from them, we could find the CUDA-based parallel GA always gains better performance with the average value and the best value of the objective function than the classical GA.

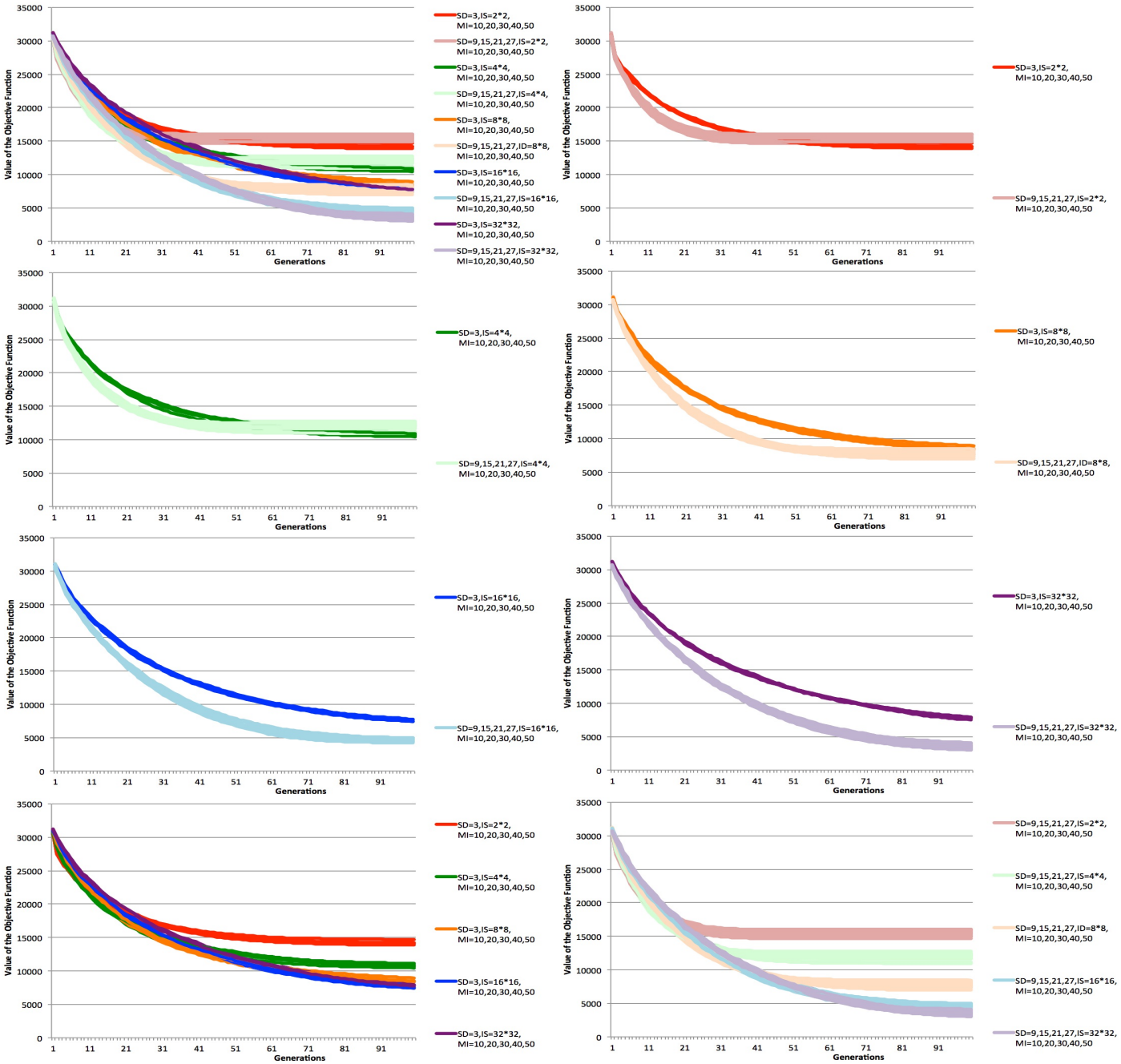

Fig. 5. Contorolling Parameters Configuration

C. *Comparison Test on Execution Time*

TABLE II. DIFFERENT IMPLEMENTATIONS TO OBTAIN THE EXECUTION TIME

Hgpu	Proposed hybrid GA over NVIDIA K40 GPU
Scpu	Classical GA over Intel Xeon E5640 CPU with one core
MSmulticpu	OpenMP based master-slave GA over Intel Xeon E5640 CPU with four cores
MSgpu	Master-slave GA over NVIDIA K40 GPU
Hcpu	Proposed hybrid GA over Intel Xeon E5640 CPU with one core
Hmulticpu	OpenMP based hybrid GA over Intel Xeon E5640 CPU with four cores

For fair comparison, we do not only use the serial CPU, but also take OpenMP based parallel CPU to contrast the execution time with GPU for implementations of the classical GA, the master-slave GA and the proposed hybrid GA separately. Different implementations used to obtain the execution time are noted in Table 2.

Since parallel implementation is one of the most promising options to accelerate GAs, parallel GAs always work faster than serial GAs as displayed in Fig. 7. Although, the CUDA-based parallel GA does not win against other parallel algorithms with the small size population, the performance has been improved dramatically by increasing this latter parameter.

Moreover, we expect that it can achieve further acceleration for more complicated problems.

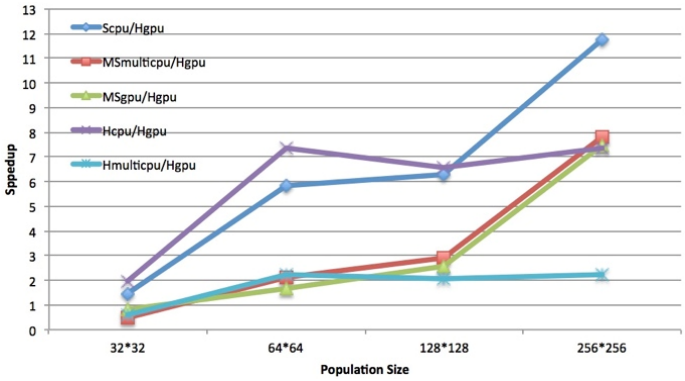

Fig. 7. Execution Time Comparison

VI. Conclusions

A parallel Genetic Algorithm consisting of an island model at the upper level and a fine-grained model at the lower level is proposed in this paper. It is highly consistent with the CUDA framework and combines metrics from two levels by keeping population diversity and increasing convergence speed. Controlling parameters configuration test witnesses that its performance is optimized with a medium size selection diameter, a relatively large island size and a wide range size migration interval. Through numerical experiments, the proposed method overcomes the classical GA by obtaining better results and taking less execution time.

Acknowledgment

This work was supported by a scholarship from the China Scholarship Council (CSC).

Appendix

Model and Constraints	Descriptions
$WT * \sum T_j + C_{max}$	Objective function
$T_j = \max\left(S_{jS} + P_{jSM_{jS}} - D_j, 0\right), j \in \{1,2,\dots,J\}$	Tardiness
$C_{max} = \max_j(S_{jS} + P_{jSM_{jS}}), j \in \{1,2,\dots,J\}$	Makespan
$S_{js} \geq S_{j\,s-1} + P_{j\,s-1\,M_{j\,s-1}}, j \in \{1,2,\dots,J\}, s \in \{2,3,\dots,S\}$	Precedence among operations at the first stage
$S_{js} \geq S_{j\,s-1} + P_{j\,s-1\,M_{j\,s-1}}, j \in \{1,2,\dots,J\}, s \in \{2,3,\dots,S\}$	Precedence among operations after the first stage
$S_{js}+P_{jsM_{js}} \leq S_{j's}, j \in \{1,2,\dots,J\}, j' \in \{1,2,\dots,J\}, s \in \{1,2,\dots,S\}, j \neq j', M_{js} == M_{j's}, S_{js} \leq S_{j's}$	Precedence caused by the sequencing on machines

References

[1] Holland, J. H. (1992). Genetic algorithms. *Scientific American*, *267*(1), 66-73.

[2] Cantú-Paz, E. (1998). A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, *10*(2), 141-171.

[3] Somani, A., & Singh, D. P. (2014, August). Parallel Genetic Algorithm for solving Job-Shop Scheduling Problem Using Topological sort. In *Advances in Engineering and Technology Research (ICAETR), 2014 International Conference on* (pp. 1-8). IEEE.

[4] Lin, S. C., Goodman, E. D., & Punch, W. F. (1997, April). Investigating parallel genetic algorithms on job shop scheduling problems. In *International Conference on Evolutionary Programming* (pp. 383-393). Springer, Berlin, Heidelberg.

[5] Park, B. J., Choi, H. R., & Kim, H. S. (2003). A hybrid genetic algorithm for the job shop scheduling problems. *Computers & industrial engineering*, *45*(4), 597-613.

[6] NVIDIA. (2017) Cuda 8.1. [Online]. https://developer.nvidia.com/cuda-toolkit

[7] Munawar, A., Wahib, M., Munetomo, M., & Akama, K. (2009). Hybrid of genetic algorithm and local search to solve MAX-SAT problem using Nvidia CUDA framework. *Genetic Programming and Evolvable Machines*, *10*(4), 391.

[8] Boyer, V., & El Baz, D. (2013, May). Recent advances on GPU computing in operations research. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International* (pp. 1778-1787). IEEE.

[9] Akhshabi, M., Haddadnia, J., & Akhshabi, M. (2012). Solving flow shop scheduling problem using a parallel genetic algorithm. *Procedia Technology*, *1*, 351-355.

[10] Tamaki, H. (1992). A paralleled genetic algorithm based on a neighborhood model and its application to the jobshop scheduling. *Parallel Problem Solving from Nature 2*, 573-582.

[11] Ghosn, S. B., Drouby, F., & Harmanani, H. M. (2016). A parallel genetic algorithm for the open-shop scheduling problem using deterministic and random moves. *Int. J. Artif. Intell*, *14*(1), 130-144.

[12] Pospichal, P., Jaros, J., & Schwarz, J. (2010, April). Parallel genetic algorithm on the CUDA architecture. In *European conference on the applications of evolutionary computation*(pp. 442-451). Springer, Berlin, Heidelberg.

[13] Zhang, S., & He, Z. (2009, October). Implementation of parallel genetic algorithm based on CUDA. In *International Symposium on Intelligence Computation and Applications* (pp. 24-30). Springer, Berlin, Heidelberg.

[14] AitZai, A., Boudhar, M., & Dabah, A. (2013). Parallel CPU and GPU computations to solve the job shop scheduling problem with blocking.

[15] Zajıcek, T., & Šucha, P. (2011). Accelerating a Flow Shop Scheduling Algorithm on the GPU. *eraerts*, 143.

[16] Bruzzone, A. A. G., Anghinolfi, D., Paolucci, M., & Tonelli, F. (2012). Energy-aware scheduling for improving manufacturing process sustainability: A mathematical model for flexible flow shops. *CIRP Annals-Manufacturing Technology*, *61*(1), 459-462.

[17] Plazolles, B., El Baz, D., Spel, M., Rivola, V., & Gegout, P. (2017). SIMD Monte-Carlo Numerical Simulations Accelerated on GPU and Xeon Phi. *International Journal of Parallel Programming*, 1-23.

[18] Kohlmorgen, U., Schmeck, H., & Haase, K. (1999). Experiences with fine-grainedparallel genetic algorithms. *Annals of Operations Research*, *90*, 203-219.