Performance Analysis of a Parallel Genetic Algorithm: A Case Study of the Traveling Salesman Problem

Henry Novianus Palit
Department of Informatics
Petra Christian University
Indonesia
hnpalit@petra.ac.id

Indar Sugiarto
Department of Electrical Eng.
Petra Christian University
Indonesia
indi@petra.ac.id

Doddy Prayogo
Department of Civil Eng.
Petra Christian University
Indonesia
prayogo@petra.ac.id

Alexander T.K. Pratomo
Department of Informatics
Petra Christian University
Indonesia
alexander.thomas@petra.ac.id

***Abstract*—Genetic Algorithm (GA) is one of the most popular optimization techniques. Inspired by the theory of evolution and natural selection, it is also famous for its simplicity and versatility. Hence, it has been applied in diverse fields and domains. However, since it involves iterative and evolutionary processes, it takes a long time to obtain optimal solutions. To improve its performance, in this research work, we had parallelized GA processes to enable searching through the solution space with concurrent efforts. We had experimented with both CPU and GPU architectures. Speed-ups of GA solutions on CPU architecture range from 7.2 to 22.2, depending on the number of processing cores in the CPU. By contrast, speed-ups of GA solutions on GPU architecture can reach up to 172.4.**

Keywords—genetic algorithm, parallel, OpenMP, CUDA, traveling salesman problem

I. Introduction

Genetic Algorithm (GA) is a metaheuristic to solve optimization and search problems by relying on biologically inspired operators such as selection, crossover, and mutation. It is inspired by the process of natural selection belonging to the larger class of evolutionary algorithms. Since its conception by John Holland and his collaborators in the 1960s and 1970s, many variants of GAs have been proposed and used to address a variety of optimization problems [1]: from graph coloring to pattern recognition, from discrete systems (e.g., the traveling salesman problem) to continuous systems (e.g., the efficient design of airfoil), and from financial analysis to multi-objective engineering optimization.

To explore the search space, GA involves multiple individuals that can randomly and independently find a mate in the population and produce new individuals. This characteristic process is excellent for parallelization, and consequently, different optimization parameters and objectives can be examined simultaneously and quickly. Several parallel schemas for GAs [2] – either on CPU, GPU, or both – have been proposed in the past decade. Leveraging the multi-processing capabilities provided by today's computers, we believe the use of parallel GAs will empower many research and industrial works to obtain feasible solutions in a short time.

As part of our efforts to build capabilities and experiences with our high throughput computing platform (PakCarik) [3], we evaluate the execution of a parallel GA – using the Traveling Salesman Problem (TSP) as the case study – on the computing platform. This manuscript is just a preliminary report on our research work, as we will explore other possibilities to speed up GA with parallelization in the future.

In the following sections, we will detail some applications of GAs and some techniques to parallelize GAs. Next, the GA implementation for solving TSP is discussed. Finally, we will present the results of this research work and discuss them, before giving conclusion.

II. Applications of Genetic Algorithms

Generally, there are two major areas of potential for GAs [4]: optimizing an operating system and fitting a quantitative model. Such examples of operating system are a gas distribution pipeline system, traffic lights, traveling salesmen, allocation of funds to projects, scheduling, handling and blending of materials, and so forth. The system designer or operator usually selects a few decision parameters (perhaps within constraints) and measure the system's performance by some relevant objective or fitness function. The other potential area, which arguably has been less explored and discussed, deals with testing and fitting quantitative models. Instead of maximizing the performance of an operating system, here we are trying to find parameters that minimize the misfit between the model and the data. The fitness function (or, more aptly called the "misfit function") represents the difference between the observed and predicted data values. Hence, the optimization objective is to obtain parameter values for the model that minimize the misfit function.

TABLE I. LEVELS OF PARALLELISM

Processing Level	Granularity	Typical Instruction#	Computer Architecture*
Job (Program) Level	Coarse	1,000,000's	MIMD (generally MPMD)
Subprogram Level	Medium – Coarse	10,000's – 100,000's	MIMD (SPMD or MPMD)
Procedure Level	Medium	< 2,000	MIMD (generally SPMD)
Loop Level	Fine	< 500	SIMD (and some MIMD)
Instruction Level	Fine	< 20	Often processor-specific, compiler-assisted

[a] Source: Summarized from Hwang and Jotwani [5]

*Note: SIMD = Single Instruction-stream, Multiple Data-streams
MIMD = Multiple Instruction-streams, Multiple Data-streams
SPMD = Single Program, Multiple Data-streams
MPMD = Multiple Programs, Multiple Data-streams

III. Parallelizing Genetic Algorithms

Noting the broad and versatile use of GAs to a vast of problems, it would really be sensible and desirable thing to speed up GA's iterative, evolutionary process. Parallel computing, the simultaneous use of multiple compute resources to solve a computational problem, is deemed suitable to address the speed-up issue. To leverage on parallel computing, the computational problem should be able to:

- Be divided into smaller segments that can be handled concurrently;
- Run multiple program instructions at any given time;
- Be executed in a shortened time with multiple compute resources.

Typical compute resources for parallel computation are a single computer with multiple processors / cores (i.e., CPUs) or an arbitrary number of such computers connected to a network. Over tens of processes (or threads) may be executed simultaneously on this multi-processor platform. Parallelism may run on different processing levels, as shown in Table 1. The lower the processing level being parallelized, the finer the granularity of the software processes, and usually the higher the parallelism gain (although the communication and scheduling overheads may offset the parallelism gain). A typical parallel program may involve a combination of these levels of parallelism [5]. The actual combination depends on the application, formulation, algorithm, language, program, compilation support, and hardware characteristics.

To run GA in a parallel fashion, the basic idea is to distribute the computations over multiple processors or computers. GA operators such as selection, crossover, and mutation are commonly implemented as procedures (subroutines) or subprograms. Leveraging the multi-processor (multi-core) platform, many research works embrace parallelism by running a sequential GA operator directly on multiple processors to exploit data parallelism. In this scenario, all computations pertaining to a particular GA operator are assigned to a single process / thread (and consequently, to a single processing core). Multiple GA operators, each of which handles a different data-stream, can run on multiple cores in a parallel fashion. Hence, from the algorithm's point of view, each instance of parallel processes is essentially a sequential GA operator. This naive approach can still yield significant speed-up on the GA processes. Nevertheless, higher speed-up may be obtained as we seek finer granularity of the parallel processes. In addition, the emergence of CPU + GPU heterogeneous architecture has attracted many researchers and practitioners to exploit this computing platform.

A. *Parallelization with OpenMP*

OpenMP [6] is an API (Application Programming Interface) comprising compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in programs written in Fortran, C, or C++. It is the most widely used standard for SMP (Symmetric Multi-Processing) systems. OpenMP provides special notation (e.g., instructions or directives) to specify how a program's fragments are assigned to the individual processors / cores, as well as to control the ordering of accesses to shared data by different threads. This information will be used by the compiler to generate the actual machine code for execution by each processor.

It is the onus of the program developer to dictate where and how parallelization should be carried out in the program. Employing the OpenMP notation, we can specify the code fragments (e.g., loop nests) of a GA process to be parallelized and later executed by multiple threads. Thus, the fine-grained parallelization (i.e., on loop level) can be attained.

B. *Parallelization with GPU + CUDA*

With the proliferating use of GPU (Graphics Processing Unit) for accelerating computations, not just limited to image or video processing, we may further gain speed-up from hundreds to thousands of cores available. GPU computing has thrusted the research of parallel GAs to the world of high-performance computing (HPC) and brought forth a great potential to many research and industrial works that can benefit from the GPU-accelerated stochastic and global search for better solutions [2]. However, as indicated earlier, many research works adopted a naive approach: running a sequential GA process on a GPU thread, in exactly the same fashion as how parallel GAs generally run on CPU. As GPU provides a massive number of processing cores, the parallel program on GPU should be designed differently from that on CPU.

Developed by NVIDIA and introduced in 2006, CUDA (Compute Unified Device Architecture) platform [7] exposes GPU memory and execution models for developers to leverage its computing power. Many CUDA libraries and tools are available to provide developers everything they need to build GPU-accelerated programs in popular languages such as C, C++, Fortran, Python, and MATLAB. Parallelism can be expressed through the given extensions in the form of a few basic keywords. The sequential parts of the GPU-accelerated program still run on CPU, whereas the compute intensive portions run simultaneously on hundreds or thousands of GPU cores.

Similar to the OpenMP program, the GPU-accelerated program can attain fine-grained parallelization through explicit programming. Different to the development of an OpenMP program that relatively requires no additional effort on the program developer, building a GPU-accelerated program requires much understanding of the GPU system architecture and the CUDA programming model.

IV. Genetic Algorithm for Traveling Salesman Problem

This section details various aspects required to solve TSP using GA. Firstly, we describe the representation of the path that the salesman needs to travel. Next, we explain how the basic GA operators are implemented for TSP.

A. *Path Representation*

Since the traveled path is essentially the solution of TSP, the path should be coded as the individual's chromosome. For example [8], a tour $1 \rightarrow 4 \rightarrow 8 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 6 \rightarrow 7$ (where each number represents a location ID) can be represented as a chromosome with sequence (1 4 8 2 5 3 6 7). Note that each location must be visited exactly once.

The traveling distance, from location 1 to location 7 in the above example, can represent the GA's fitness value. Since the best solution should have the shortest traveling distance, the iterative GA processes will try to minimize the fitness value as low as possible.

B. *Selection Operator*

The purpose of selection operator is to select some individuals from the population for later breeding (i.e., crossover operator). The selection operator is expected to have high probability to find the good individuals (i.e., good solutions), so they as parents will produce other good, or even better, offspring. The individual's fitness value can differentiate the good from the bad. Since the fitness value in TSP is represented by the traveling distance, the less is the better: minimization problem.

Two methods are employed for the selection operator:

- Tournament Selection: Running several tournaments among some individuals selected randomly from the population. In each tournament, the individual with the best fitness (i.e., the least value) is the winning candidate for generating offspring later.
- Roulette Wheel Selection (a.k.a. Fitness Proportionate Selection): A sector of the wheel is proportionately (based on the reciprocal fitness value) assigned to each individual. The winning candidate is then selected by rotating the wheel (i.e., randomly picked).

C. *Crossover Operator*

The classical crossover operators such as one-point, two-point, and uniform crossovers would not be appropriate for TSP, since the results should maintain the combinatorial nature of sequencing locations. Alternatively, the partially mapped, order, and cycle crossover operators were mostly suggested in past research works [8]. Among those three crossover operators, we employ the order one, as maintaining the relative order of locations is quite important in our opinion. The order crossover operator is explained in the following paragraph.

Consider the chromosomes of two parents to be crossover-ed are as follows (with randomly two cut points marked by "|"):

P1 = (3 4 8 | 2 7 1 | 6 5)
P2 = (4 2 5 | 1 6 8 | 3 7)

The offspring should maintain the parents' sequences within the two cut points, which gives

O1 = (x x x | 2 7 1 | x x)
O2 = (x x x | 1 6 8 | x x)

Afterward, starting from the second cut point of one parent, the sequence of locations from the other parent is copied in the same order omitting existing locations. For instance, the sequence of locations in the second parent from the second cut point is 3→7→4→2→5→1→6→8. After omitting locations 2, 7, and 1 (which has existed in the first offspring), the resulting sequence is 3→4→5→6→8, which is then placed in the first offspring starting from the second cut point:

O1 = (5 6 8 | 2 7 1 | 3 4)
O2 = (4 2 7 | 1 6 8 | 5 3)

The second offspring is completed in the same way.

D. *Mutation Operator*

Mutation operator is used to maintain genetic diversity in the population from one generation to the next. It is also an attempt to avoid being trapped in the local optima. Every individual is subject to mutation, although mutation seldom occurs (i.e., very small probability). When an individual is selected for mutation, two genes of its chromosome would be randomly determined and then exchanged.

E. *Elitism Operator*

Elitism operator allows a few best individuals (i.e., having the least fitness values) from the current generation to carry over to the next, unaltered. To a certain extent, this would guarantee that the GA solution quality does not degrade from one generation to another. The proportion of elites should be kept small to maintain diversity and avoid premature convergence.

V. Results and Discussion

This section begins with hotspot analysis on the sequential GA program. Next, we detail the test environment for our research work. The main findings of this research work are then presented and discussed.

A. *Hotspot Analysis on Sequential GA Program*

VTune Profiler (formerly VTune Amplifier) [9] was employed for this hotspot analysis. Displayed in Table 2 are the top 10 subroutines having the longest running time in the sequential GA program. These subroutines (except those from the standard library) are our target for parallelization. As predicted, the crossover operator (involving subroutines starting with XO) took the lion's share of the execution time.

TABLE II. Hotspot Analysis

Subroutine Name	Running Time (in minutes)
XOCross	75.37
Mutation	72.99
CalculateGenesFitness	62.81
XORankedPairing	31.19
XOCutting	28.42
XOGeneChecking	21.91
SelectionElitism	20.62
GenerateGene	4.03
std::stable_sort	1.14
XOJoinGene	0.87

B. *Test Environment*

This subsection explains the TSP test cases to be solved with GA and the compute resources employed for running different (i.e., three variants) GA programs.

1) *TSP Test Cases*

In our experiment, the TSP test cases were downloaded from the National Traveling Salesman Problems webpage [10], maintained by the Department of Combinatorics and Optimization, the University of Waterloo (Canada). In particular, three test cases were used:

- CA4663: a list of 4,663 cities in Canada (the optimal tour has the length of 1,290,319);
- FI10639: a list of 10,639 cities in Finland (the optimal tour has the length of 520,527);
- IT16862: a list of 16,862 cities in Italy (the optimal tour has the length of 557,315).

2) *Compute Resources*

Two computers were mainly used for running different GA programs to solve TSP test cases. The third computer (with the highest specification) was only used for the most complex test case (i.e., IT16862). Below are the specifications of the computers:

- Comp1 (*R5-1600*, *RTX 2060*)
  CPU : AMD Ryzen 5 1600 (6 cores, 3.2 GHz)
  RAM : 16 GB DDR4 (3200 MHz)
  GPU : NVIDIA GeForce RTX 2060 (30 SM x 64 CUDA cores, 1365 MHz, 6 GB GDDR6)
- Comp2 (*E5-2630v4*, *Tesla P4*)
  CPU : 2 (two) Intel Xeon E5-2630 v4 (10 cores, 2.2 GHz)
  RAM : 128 GB DDR4 (2133 MHz)
  GPU : NVIDIA Tesla P4 (20 SM x 128 CUDA cores, 886 MHz, 8 GB GDDR5)
- Comp3 (*TR-2990WX*, *RTX 2080 Ti*, *GTX 1070 Ti*)
  CPU : AMD Ryzen Threadripper 2990 WX (32 cores, 3.0 GHz)
  RAM : 64 GB DDR4 (2400 MHz)
  GPU : NVIDIA GeForce RTX 2080 Ti (68 SM x 64 CUDA cores, 1350 MHz, 11 GB GDDR6) & NVIDIA GeForce GTX 1070 Ti (19 SM x 128 CUDA cores, 1607 MHz, 8 GB GDDR5)

3) *GA Programs*

We developed 3 (three) variants of GA to solve TSP cases: Sequential GA, Parallel (OpenMP) GA, and Parallel (CUDA) GA. Other GA parameters that we set are:

- Mutation Rate: 5%,
- Elitism Rate: 5%,
- Population Size: number of cities * 0.25,
- Tournament Size: 3.

C. *Comparative Results*

For each TSP test case, the three variants of GA were run on the test computers, and results (i.e., average number of generations per minute) were compared.

1) *Test Case CA4663*

As shown in Table 3, the parallel GA solutions can produce more generations per minute than the sequential GA solutions do. For CPU architecture, the AMD Ryzen 5 1600 obtains the speed-up of 7.4x when it is executed in parallel (with OpenMP), whereas the Intel Xeon E5-2630 v4 obtains the speed-up of 14.8x when similarly executed in parallel (with OpenMP).

For GPU architecture, the NVIDIA GeForce RTX 2060 obtains the speed-up of 82.9x over the AMD Ryzen 5 1600 (both GA solutions, CPU sequential and CUDA parallel, were executed in the same Comp1 machine). The NVIDIA Tesla P4 obtains the speed-up of 55.9x over the Intel Xeon E5-2630 v4 (both GA solutions, CPU sequential and CUDA parallel, were executed in the same Comp2 machine).

Observing the fitness value progress in Figure 1, we see that the sequential GA solutions took very long time to converge, and even after 12-hour run, they were still far away from the optimal solution. On the other hand, the parallel (OpenMP) GA solutions cut down the individuals' fitness values steeply in the first 1 hour (60 minutes), and afterward, the fitness values were reduced gradually and steadily. The parallel (CUDA) GA solutions cut down the individuals' fitness values steeply in the first 30-40 minutes, and then, the fitness values were reduced very slowly. After the 12-hour run, the best fitness value (with the length of 1,527,875.8) is about 15.5% less optimal than the most optimal tour (with the length of 1,290,319).

TABLE III. COMPARATIVE PERFORMANCE OF GA SOLUTIONS FOR TEST CASE CA4663 (AFTER 12-HOUR RUN)

GA Solution	Avg. Generation# per Minute	Min. Fitness	Avg. Fitness	Max. Fitness
E5-2630v4S* (CPU)	63.7	4,372,899.0	4,468,292.1	4,835,286.1
R5-1600S* (CPU)	81.8	3,673,959.5	3,756,221.1	4,117,880.8
R5-1600 (OpenMP)	603.4	2,111,284.3	2,268,885.6	2,732,384.0
E5-2630v4 (OpenMP)	940.0	1,967,163.1	2,127,867.7	2,510,704.3
Tesla P4 (CUDA)	3,561.7	1,564,348.6	1,656,947.6	1,822,434.8
RTX 2060 (CUDA)	6,781.3	1,527,875.8	1,685,845.1	1,892,503.0

*Note: The GA solution ending with 'S' means the sequential variant

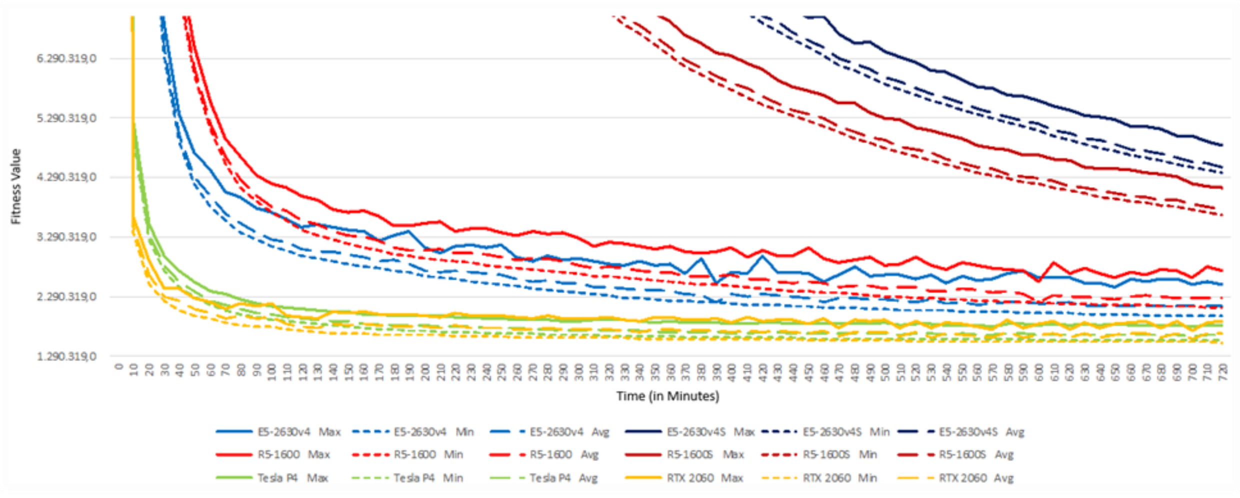

Fig. 1. Fitness Value Progress of GA Solutions for Test Case CA4663 (after 12-hour run)

TABLE IV. COMPARATIVE PERFORMANCE OF GA SOLUTIONS FOR TEST CASE FI10639 (AFTER 12-HOUR RUN)

GA Solution	Avg. Generation# per Minute	Min. Fitness	Avg. Fitness	Max. Fitness
E5-2630v4S* (CPU)	12.0	15,752,507.8	15,808,694.2	15,889,584.2
R5-1600S* (CPU)	15.0	14,154,926.6	14,194,222.1	14,252,572.3
R5-1600 (OpenMP)	109.5	3,030,184.6	3,052,983.9	3,116,257.0
E5-2630v4 (OpenMP)	192.0	1,910,552.7	1,931,680.6	1,995,457.2
Tesla P4 (CUDA)	476.2	1,096,557.4	1,116,583.2	1,145,332.1
RTX 2060 (CUDA)	1,224.7	888,951.5	903,758.6	927,015.0

*Note: The GA solution ending with 'S' means the sequential variant

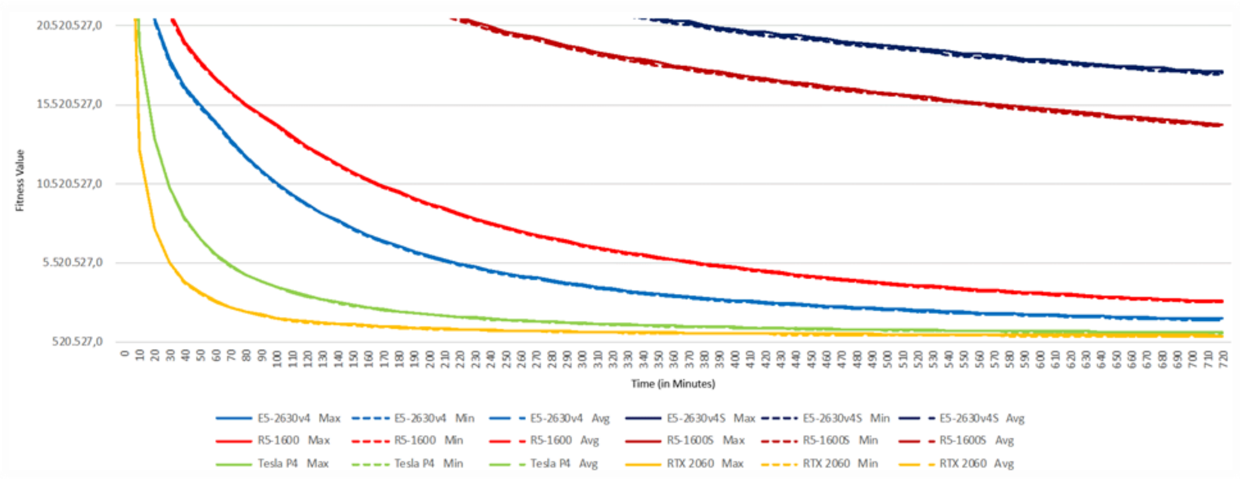

Fig. 2. Fitness Value Progress of GA Solutions for Test Case FI10639 (after 12-hour run)

2) *Test Case FI10639*

Table 4 shows that the parallel GA solutions can produce more generations per minute than the sequential GA solutions do. For CPU architecture, the AMD Ryzen 5 1600 obtains the speed-up of 7.3x when it is executed in parallel (with OpenMP), whereas the Intel Xeon E5-2630 v4 obtains the speed-up of 16.0x when similarly executed in parallel (with OpenMP). Overall, the speed-ups we obtain for this test case (FI10639) are pretty similar to those we obtain for the previous test case (CA4663).

For GPU architecture, the NVIDIA GeForce RTX 2060 obtains the speed-up of 81.6x over the sequential solution running on the AMD Ryzen 5 1600. The NVIDIA Tesla P4 obtains the speed-up of 39.7x over the sequential solution running on the Intel Xeon E5-2630 v4. The speed-up on NVIDIA GeForce RTX 2060 we obtain for this test case (FI10639) is quite similar to that we obtain for the previous test case (CA4663). By contrast, there is a drop in speed-up on NVIDIA Tesla P4 (from 55.9 to 39.7) when we increase the test case's complexity (from CA4663 to FI10639).

Observing the fitness value progress in Figure 2, we see that the sequential GA solutions were still far away from the optimal solution even after 12-hour run. The parallel (OpenMP) GA solutions cut down the individuals' fitness values steeply in the first 1 hour, and afterward, the fitness values were reduced gradually and steadily. The parallel (CUDA) GA solutions cut down the individuals' fitness values steeply in the first 30-40 minutes, and then, the fitness values were reduced very slowly. After the 12-hour run, the best fitness value is still 41.4% less optimal than the most optimal tour (with the length of 520,527).

3) *Test Case IT16862*

Table 5 shows that the parallel GA solutions can produce more generations per minute than the sequential GA solutions do. For CPU architecture, the AMD Ryzen 5 1600, the Intel Xeon E5-2630 v4, and the AMD Ryzen Threadripper 2990 WX obtain the speed-ups of 7.2x, 15.9x, and 22.2x, respectively, when they are executed in parallel (with OpenMP). Overall, the speed-ups we obtain on the AMD Ryzen 5 1600 and the Intel Xeon E5-2630 v4 for this test case (IT16862) are pretty similar to those we obtain on the respective computers for the previous test cases (CA4663 and FI10639).

For GPU architecture, the NVIDIA GeForce RTX 2060 obtains the speed-up of 59.0x over the sequential solution running on the AMD Ryzen 5 1600. The NVIDIA Tesla P4 obtains the speed-up of 22.8x over the sequential solution running on the Intel Xeon E5-2630 v4. Finally, the NVIDIA GeForce RTX 2080 Ti and the NVIDIA GeForce GTX 1070 Ti obtain the speed-ups of 172.4x and 42.7x, respectively, over the sequential solution running on the AMD Ryzen Threadripper 2990 WX. There are significant drops in speed-ups on NVIDIA Tesla P4 (from 55.9 to 22.8) and NVIDIA GeForce RTX 2060 (from 82.9 to 59.0), when we increase the test case's complexity (from CA4663 to IT16862). It is obvious that those GPU devices were no longer scalable. Among the GPU devices, the NVIDIA GeForce RTX 2080 Ti yields the best speed-up (172.4).

Observing the fitness value progress in Figure 3, we see that the sequential GA solutions were still far away from the optimal solution even after 12-hour run. The parallel (OpenMP) GA solutions cut down the individuals' fitness values steeply in the first 1 hour, and afterward, the fitness values were reduced gradually and steadily. The parallel (CUDA) GA solutions cut down the individuals' fitness values steeply in the first 30-40 minutes, and then, the fitness values were reduced very slowly. After the 12-hour run, the best fitness value is still 53.0% less optimal than the most optimal tour (with the length of 557,315).

TABLE V. COMPARATIVE PERFORMANCE OF GA SOLUTIONS FOR TEST CASE IT16862 (AFTER 12-HOUR RUN)

GA Solution	Avg. Generation# per Minute	Min. Fitness	Avg. Fitness	Max. Fitness
E5-2630v4S* (CPU)	4.9	35,162,753.9	35,291,258.7	35,463,585.1
TR-2990WXS* (CPU)	5.1	34,423,275.3	34,549,554.5	34,706,707.7
R5-1600S* (CPU)	5.7	33,315,569.9	33,451,954.6	33,622,007.5
R5-1600 (OpenMP)	41.0	13,966,778.9	14,006,352.0	14,091,529.7
E5-2630v4 (OpenMP)	78.0	7,750,217.7	7,774,997.7	7,846,569.6
TR-2990WX (OpenMP)	113.1	5,289,105.1	5,314,011.7	5,382,960.2
Tesla P4 (CUDA)	111.8	4,178,787.6	4,199,267.2	4,228,963.7
GTX 1070 Ti (CUDA)	217.7	2,501,517.8	2,518,557.8	2,545,618.7
RTX 2060 (CUDA)	336.3	1,900,081.5	1,921,036.1	1,949,108.7
RTX 2080 Ti (CUDA)	879.0	1,185,113.7	1,208,297.9	1,238,566.8

*Note: The GA solution ending with 'S' means the sequential variant

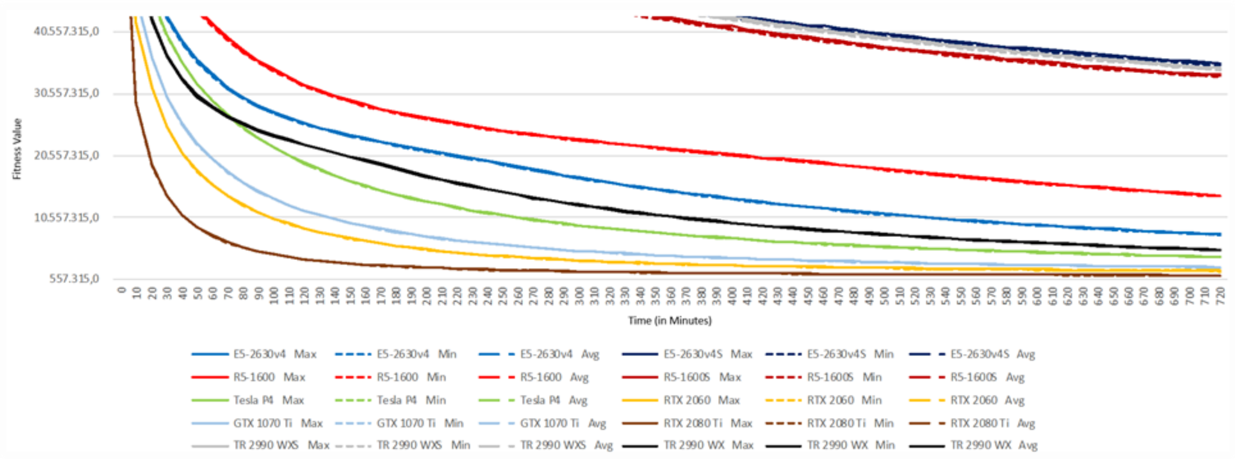

Fig. 3. Fitness Value Progress of GA Solutions for Test Case IT16862 (after 12-hour run)

VI. CONCLUSION AND FUTURE WORK

Owing to its simplicity and versatility, genetic algorithm (GA) has been applied to diverse fields. With the rising use of GPU in many computation platforms, we can leverage on its huge number of cores to accelerate processes. GA's characteristics – which involve many individuals, comprise some operators, and run iteratively in search for optimal solutions – can gain much benefit from GPU acceleration.

We had parallelized some GA operators using OpenMP (for running on CPU) and CUDA (for running on GPU). The traveling salesman program (TSP) was employed as the case study. Our experimental results show that indeed we can obtain significant speed-ups with both devices. Speed-ups of GA solutions on CPU range from 7.2 to 22.2, depending on the number of processing cores in the CPU. By contrast, speed-ups of GA solutions on GPU can reach up to 172.4.

In the near future, we want to employ different parallelization techniques for GA to get more benefits from multiple GPU devices. We also intend to improve our understanding and capability in this CPU-GPU architecture by testing out different but equally challenging algorithms.

ACKNOWLEDGMENT

This research project was partially supported by the Indonesian Ministry of Education, Culture, Research and Technology / National Research and Innovation Agency through the PDUPT Grant No. 007/SP2H/PDUPT/LPPM-UKP/IV/2021 and also by the Bureau of Research and Community Service as well as the Faculty of Industrial Technology at Petra Christian University through the research funding: No. 01/PNLT/FTI/UKP/2018. We express our sincere gratitude and appreciation for all the supports.

REFERENCES

[1] X.-S. Yang, "Genetic Algorithms," in Nature-Inspired Optimization Algorithms, 2nd ed., Academic Press, 2021, pp. 91-100.

[2] J. R. Cheng and M. Gen, "Parallel Genetic Algorithms with GPU Computing," IntechOpen, 5 February 2020. [Online] Available: https://www.intechopen.com/chapters/69121.

[3] I. Sugiarto, D. Prayogo, H. Palit, F. Pasila, R. Lim, A. Noertjahyana, I. G. A. Widyadana, S. Hermawan, A. B. Gumelar and B. N. Yahya, "Custom Built of Smart Computing Platform for Supporting Optimization Methods and Artificial Intelligence Research – PakCarik: GPU-Accelerated Platform for AI Researches," Proceedings of the Pakistan Academy of Sciences: A. Physical and Computational Sciences, vol. 58, no. S, pp. 59-64, 2021.

[4] J. Everett, "Model Building, Model Testing and Model Fitting," in The Practical Handbook of Genetic Algorithms, Applications, 2nd ed., Boca Raton (FL, USA), Chapman & Hall / CRC, 2001, pp. 1-29.

[5] K. Hwang and N. Jotwani, Advanced Computer Architecture: Parallelism, Scalability, Programmability, 3rd ed., New Delhi: McGraw-Hill Education, 2016.

[6] OpenMP Architecture Review Board, "OpenMP FAQ," 6 June 2018. [Online] Available: https://www.openmp.org/about/openmp-faq.

[7] NVIDIA, "CUDA Zone," July 2021. [Online] Available: https://developer.nvidia.com/cuda-zone.

[8] A. Hussain, Y. S. Muhammad, M. N. Sajid, I. Hussain, A. M. Shoukry and S. Gani, "Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator," Computational Intelligence and Neuroscience, vol. 2017, 2017.

[9] Intel Corporation, "Intel VTune Profiler User Guide," 9 November 2021. [Online] Available: https://www.intel.com/content/www/us/en/develop/documentation/vtune-help/top.html.

[10] W. Cook, "National Traveling Salesman Problems," 10 March 2017. [Online] Available: http://www.math.uwaterloo.ca/tsp/world/countries.html.