Traveling Salesman Problem Solver using Genetics Algorithm on CUDA Framework

Choopan Rattanapoka[1,*]

Abstract

This paper presents a design and implementation of a traveling salesman problem solver, which is a foundation problem in artificial intelligence field, program by using parallel genetics algorithm on graphical processing unit (GPU), Nvidia GTX560, via CUDA framework and compares the computing time between this implementation and a traditional genetic algorithm implementation computing by the central processing unit (CPU), Core i5-750. From the experiments, we found that the computation time using the solve traveling salesman problems on GPU is faster than on CPU around 4 times.

Keywords : Traveling Salesman Problem, Genetics Algorithm, CUDA, Java

[1] Department of Electronics Engineering Technology, College of Industrial Technology, King Mongkut's University of Technology North Bangkok.

[*] Corresponding author, E-mail: choopanr@kmutnb.ac.th

1. Introduction

The Traveling Salesman Problem (TSP) is one of the most widely discussed problems in combinatorial optimization and it is also a foundation problem in artificial intelligence field. The problem involves a traveling salesman that wants to visit each of a set of cities exactly once, starting from and returning to his home city. TSP demands high computation time especially when the number of cities becomes larger.

A Genetic Algorithm (GA) is a simple and elegant heuristic for which many researchers, for example in [1] and [2], used to find a solution of TSP in reasonable amounts of time. However, the basic genetic algorithm was designed to work in sequential fashion. Thus, when GAs is applied to harder and bigger problems, there is an increase in the time required to find adequate solutions. As a consequences, there have been multiple efforts to make GAs faster, and one of the most promising choices is to use parallel implementation such as in [3] and [4].

Currently, modern Graphic Processing Units (GPU) can be seen as low cost and very fast highly parallel general-purpose computing units due to the ability of these devices that can solve parallelizable problems much faster than traditional sequential processors. Moreover, several general purpose high-level languages for GPUs have become available such as CUDA and OpenCL and thus developers can easily develop applications using GPU. There are many researches and business products that use GPU to accelerate the computation time of the program such as in [5] and [6].

In this paper, we presents a design and implementation of a traveling salesman problem solver using parallel genetic algorithm on GPU via CUDA framework and compare the computation time used to find the solution and the result of solution with a traditional genetic algorithm on CPU.

2. Parallel Genetic Algorithm

A genetic algorithm has several distinctive features borrowed from an abstract formulation of biological systems. First, a representation of problem solutions called *genetic encoding* must be found which can be manipulated through some sort of genetic operations (*crossover*, and *mutation)* to yield other candidate solutions to the problem. Second, acting on an initial population, these transformations create the next *generation* of candidate solutions. Third, calculation of the objective function for each candidate solution supplies a measure of *fitness* which affects its likelihood of leaving surviving offspring in the next generation.

In [7], Erick Cantú-Paz presents a survey of parallel genetic algorithm which can be categorized into three main types: global single-population master-slave, single-population fine-grained, and multiple-population coarse-grained. Single-population fine-grained GAs are suited for massively parallel computers which also suit to our needs for using in GPU. The algorithm that we use to implement our program called *Cellular Genetic Algorithm*.

In a Cellular Genetic Algorithm, the population is usually structured in a regular grid of d dimension (d = 1, 2, 3) and the concept of small *neighborhood* is intensively used; this means that an individual may only interact with its nearby neighbors in the crossover step. The overlapped small neighborhoods helps in exploring the search space because the induced slow diffusion of solutions through the population provides a kind of exploration, while exploitation takes place inside each neighborhood by genetic operations.

3. Design and Implementation

In this paper, we use jCUDA which is a java bindings for CUDA, parallel programming paradigm by NVIDIA, to develop our TSP solver.

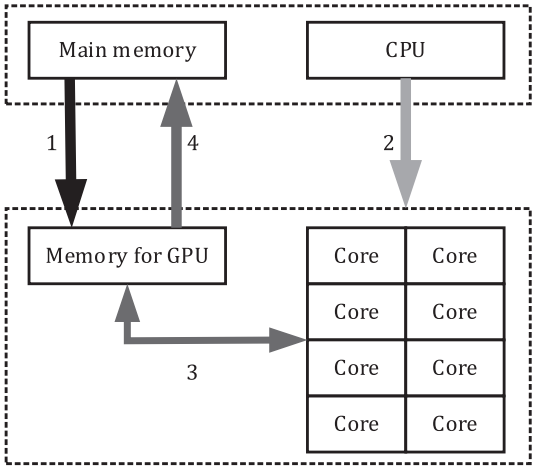

Fig. 1. Processing flow on CUDA

The GPU is especially well-suited to address problems that can be expressed as data-parallel computation (the same program is executed on many data elements in parallel). Hence, multi-thread programs can be speed up the computation when running on GPU.

The relation between CPU and GPU (processing flow on CUDA) has shown in Fig. 1. There are four main steps: 1) Copy data from main memory to GPU memory. 2) CPU instructs the process to GPU (running a GPU kernel program). 3) GPU executes program in parallel in each core also read data from and write data back to GPU memory. 4) Copy the result from GPU memory to main memory.

In our implementation, each individual (chromosome) is map on each GPU core. We also setup the layout of GPU core to 2 dimension like the layout of a 2 dimensional cellular genetic algorithm. Our cellular genetic algorithm has 4 steps: population initialization, crossover, mutation, and selection.

3.1 Population initialization

In this step, the individual solutions are randomly generated to form an initial population by CPU. Each chromosome contains the order and position of cities in the tour.

Then, CPU copies the generated data from main memory to GPU memory and instructs the process to GPU via kernel program. Thus, each GPU core will map on a different chromosome data in GPU memory.

3.2 Crossover

The crossover is a process of taking more than one chromosome (parent solutions) and producing offspring (child solutions) from them. Because we use cellular genetic algorithm, so each GPU core interacts with only its four neighbors (in Fig 2.) to do the crossover operation.

In our implementation, we use the partially matched crossover (PMX) operator [8] which has two crossover points selected randomly from the parent's chromosomes to produce the offspring. The two crossover points give a matching selection which is used to affect a cross through position by position exchange operations. The algorithm can be realized through four steps:

For example, consider two parents (P_1 and P_2):

P_1: 1 2 3 | 4 5 6 | 7 8 9

P_2: 1 5 6 | 8 2 9 | 3 7 4

First step: Generate two crossover points randomly. Here are at the 4th and 7th position.

Second step: Exchange genes in crossover region. The result are I_1 and I_2

I_1: *1 5 6* | 4 5 6 | *3 7 4*

I_2: *1 2 3* | 8 2 9 | *7 8 9*

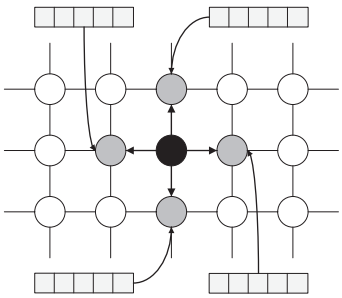

Fig. 2. Interaction between a GPU core and its neighbors in cellular genetic algorithm

Third step: Find the corresponding exchange rules in match region. Here are 4 ↔ 8, 5 ↔ 2, and 6 ↔ 9.

Fourth step: According to corresponding exchange rules in match region, genes in crossover region are exchanged. The result are C_1 and C_2 offspring.

C_1: 1 *2* *9* | 4 5 6 | 3 7 *8*

C_2: 1 *5* 3 | 8 2 9 | 7 *4* *6*

Thus, in the end of this step, each GPU core produces eight offspring.

3.3 Mutation

Mutation is a genetic operator used to make genetic diversity from one generation of a population of genetic algorithm chromosome to the next. The probability of mutation should be set low, so it means that not all of offspring has been mutated.

In our implementation, if the mutation occurs in a chromosome, we simply random two positions and then swap two genes of the chromosome in that position.

For example, the chromosome C

C: 1 2 *9* 4 5 6 *3* 7 8

Generate two positions randomly. Here are at the 3rd and 7th position. Then we obtain a new offspring C'

C': 1 2 *3* 4 5 6 *9* 7 8

3.4 Selection

After each GPU core applies genetic operators on its chromosome and obtains new eight offspring, we need to choose the best chromosome to survive for the next generation. We use *the distance of the tour* (Euclidian distance) as the fitness function to measure the quality of chromosomes. It means that the best chromosome has the shortest distance of the tour. We compare this fitness between the parent chromosome and its eight offspring and keep only the best of nine chromosomes to survive to the next generation.

Then, the genetic algorithm repeats the process of crossover, mutation, and selection until it reaches to the given number of generation.

4. Experiments

We do the experiments by solving TSP problems based on traditional genetic algorithm using CPU and cellular genetic algorithm based on GPU. Our test bed machine has Core i5-750 (8MB of cache, 2.66 GHz) CPU, 4096 MB of RAM, and NVIDIA Geforce GTX560. In our experiments, we measure the computation speed and the TSP's solution of both genetic algorithm running on CPU and GPU by adjusting the number of cities, the number of initial population and the number of generations.

4.1 Experiment 1

In this experiment, we fix the number of generations to 1000 and the number of initial population to 1024. Then, we vary the number of cities to 10, 20, 30, 40, and 50 to measure the computation speed of both genetic algorithm on CPU and GPU. From the experiment, the computation time (in Fig. 3) of the genetic algorithm run on GPU takes almost 4 times less to solve the problem than genetic algorithm run on CPU. However, the genetic algorithm on CPU has better solution than our genetic algorithm on GPU (in Fig. 4).

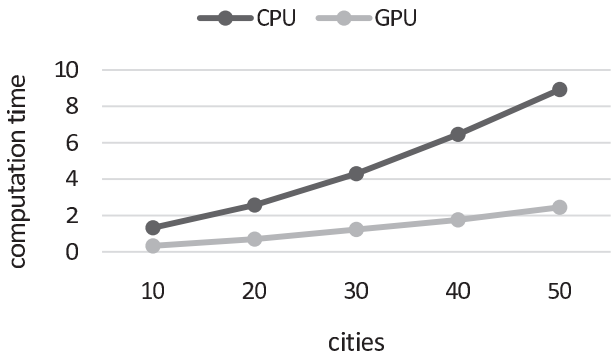

Fig. 3. The computation time of genetic algorithm run on CPU and GPU while adjusting the number of cities.

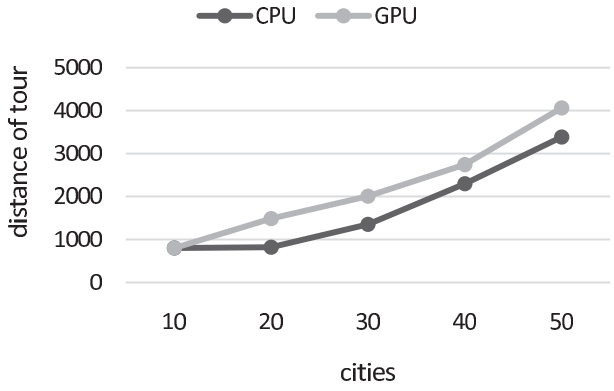

Fig. 4. The result of genetic algorithm run on CPU and GPU while adjusting the number of cities.

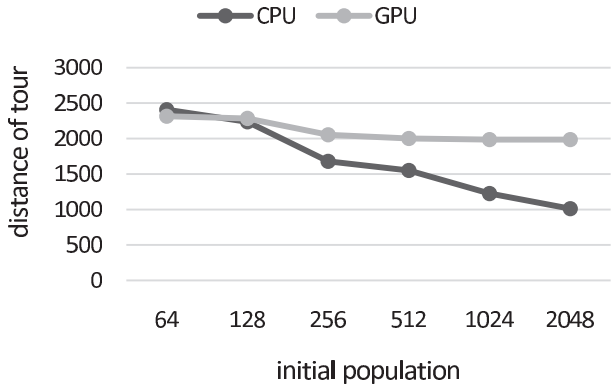

Fig. 6. The result of genetic algorithm run on CPU and GPU while adjusting the number of initial population.

4.2 Experiment 2

In this experiment, we fix the number of generations to 1000 and the number of cities to 30. Then, we vary the number of initial population to 64, 128, 256, 512, 1024, and 2048 to measure the computation speed of both genetic algorithm on CPU and GPU. From the experiment, the computation time (in Fig. 5) of the genetic algorithm run on GPU takes almost 5 times less to solve the problem than the one on CPU. However, the genetic algorithm on CPU has better solution than the one on GPU (in Fig. 6).

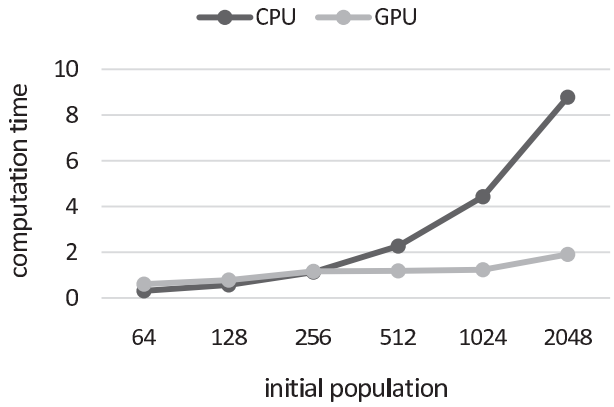

Fig. 5. The computation time of genetic algorithm run on CPU and GPU while adjusting the number of initial population.

4.3 Experiment 3

In this experiment, we fix the number cities to 30 and the number of initial population to 1024. Then, we vary the number of generations to 250, 500, 750, 1250, and 1500 to measure the computation speed of both genetic algorithm on CPU and GPU. From the experiment, the computation time (in Fig. 7) of the genetic algorithm run on GPU takes almost 4 times less to solve the problem than the one on CPU. However, the genetic algorithm on CPU has better solution than the one on GPU (in Fig. 8).

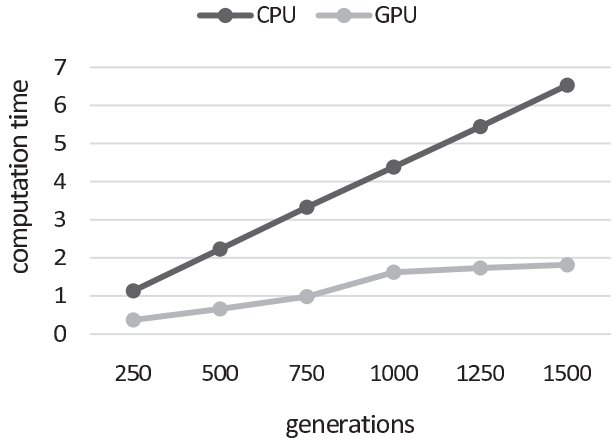

Fig. 7. The computation time of genetic algorithm run on CPU and GPU while adjusting the number of generations.

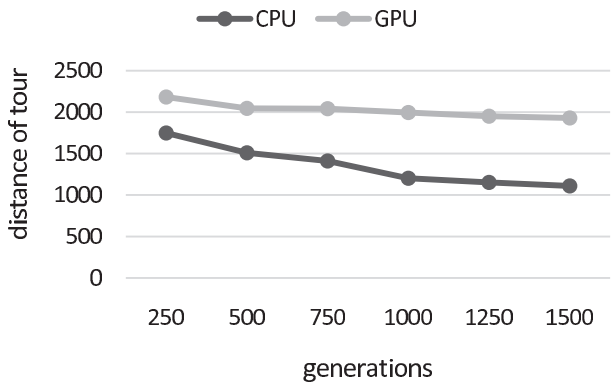

Fig. 8. The result of genetic algorithm run on CPU and GPU while adjusting the number of generations.

5. Conclusion

This paper presents a design and implementation of a traveling salesman problem solver by using cellular genetic algorithm model running on GPU via jCUDA, a java bindings for CUDA. From the experiments, we found that the computation time to solve the problem on GPU takes around four times less than the CPU's one. Especially, when we have a large number of initial population. The parallelism structure in GPU helps a lot in gaining speed of the computation. However, the result obtained from GPU is not as good as the result obtained from CPU. However, we think that the result can be better by 1) replacing our random function which is used a lot in genetic algorithm with the one providing in CUDA and 2) using other crossover operators such as 2-Opt, and 3-Opt.

6. Acknowledgements

The author would like to thank Krittin Raethong, Weerapong Kaewchooduang, and Jarong Vutthiyotin, students in the department of Electronics Engineering Technology, King Monkut's University of Technology North Bangkok that help implementing the program.

7. Reference

[1] Sangit Chatterjee, Cecilia Carrera and Lucy A. Lynch, "Genetic algorithms and traveling salesman problems", European Journal of Operational Research, vol. 93, 1996, pp. 490-510.

[2] M. Fatih Tasgetiren, P.N. Suganthan, Quan-ke Pan and Yun-Chia Liang, "A genetic algorithm for the generalized traveling salesman problem", IEEE Congress on Evolutionary Computation (CEC 2007), 2007, pp. 2382-2389.

[3] Xue Shengjun, Guo Shaoyong and Bai Dongling, "The Analysis and Research of Parallel Genetic Algorithm", Proceeding of the 4th International Conference on Wireless Communications, Networking and Mobile Computing, China, 2008, pp 1-4.

[4] S. Yussof, R.A. Razali, Ong Hang See, A.A. Ghaper and M.M. Din, "A Coarse-Grained Parallel Genetic Algorithm with Migration for Shortest Path Routing Problem", Proceeding of the 11th IEEE International Conference on High Performance Computing and Communications, South Korea, 2009, pp. 615-621.

[5] N.P. Karunadasa and D.N. Ranasinghe, "Accelerating high performance applications with CUDA and MPI", Proceeding of the 4th IEEE International Conference on Industrial and Information Systems, Sri Lanka, 2009, pp. 331-336

[6] Haiyang Zhang and F. Martin, "CUDA Accelerated Robot Localization and Mapping", Proceeding of the 5th IEEE International Conference on Technologies for Practical Robot Applications, USA, 2013, pp. 1-6.

[7] Erick Cantú-Paz, "A Survey of Parallel Genetic Algorithms", Calculateurs Paralleles, vol. 10, 1998.

[8] D.E. Goldberg and R. Lingle, "Alleles, Loci and the TSP", Proceeding of the 1st International Conference on Genetic Algorithms, 1985.