Parallel Genetic Algorithm to Optimize Travelling Salesman Problem

IT301 Course Project End-Semester Report

Saumya Karri
181ME273

Anshuman Sinha
181EE209

Sathvika B. Mahesh
181CV141

1. ABSTRACT

The project aims at investigating the efficiency of the parallel computation on the Travelling Salesman Problem using Genetic Algorithm. Performance estimation and parallelism profiling will be attempted based on OpenMP-based parallel program techniques.

It is difficult to efficiently find the solution of TSP even with an enormous number of gene instances. Evolutionary approaches including the genetic algorithm, have been widely applied in research domains to tap into the enormous potential of TSP. In this project, we propose an improved constructive crossover for TSP. We will use the algorithm to obtain optimal and/or suboptimal solutions to the popular Traveling Salesman Problem. The proposed plan is to implement it using the Parallel Virtual Machine (PVM) library, which is based on a distributed-memory approach.

2. INTRODUCTION

The **travelling salesman problem** is a combinatorial optimization problem that has vast applications. According to the problem, a salesman is required to sell his products by visiting a given set of cities. The distances between each pair of cities are provided. Therefore, to optimize his time, he should cover all the cities exactly once before returning to the original city by using the shortest possible route. This can be expressed in Computer Science/ Graph Theory terms where we have to find the lowest cost of a complete path that crosses all the nodes exactly once and returns to the starting node.

TSP Algorithm:

1. Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point.
2. Generate all permutations of cities.
3. Calculate the cost of every permutation and keep track of the minimum cost permutation.
4. Return the permutation with minimum cost.

Some Applications of TSP:

1. **Drilling of printed circuit boards:** To connect conductors of different layers, or to position the pins of integrated circuits, holes of different sizes have to drilled on the board. This can be viewed as a TSP, the aim being to minimize the travel time for the drilling machine head.
2. **Overhauling gas turbine engines:** To ensure uniform gas flow through turbines, guide vanes are provided. The problem of placing the vanes in the best possible position to get substantial benefits can be treated as a TSP.
3. **X-Ray crystallography:** An X-ray diffractometer is used to obtain information about the structure of the crystalline material. Thousands of positions of detectors are needed to get results for some experiments. To minimize the total time needed for the experiment, this is treated as a TSP.

4. **Order picking problem in warehouses:** A vehicle has to collect an order of items from a warehouse and ship it to the required destinations in least possible time. This is a classic application of TSP.
5. **Mailbox problem:** In a city, say 'n' mailboxes have to be emptied every day in a certain period of time. The problem is to find the minimum number of trucks to do this and to find the shortest time to do the collections.

Genetic algorithms are algorithms, which use concepts of evolutionary biology like recombination, inheritance, mutation, crossover etc. The algorithm replicates Darwin's process of natural selection to carry generation, i.e., survival of the fittest. The main goal of genetic algorithms is to search a solution space imitating the biological metaphor and, therefore, they are inherently parallel. Crossover and mutation play a significant role here.

The algorithm is divided into five stages primarily. Firstly, an initial population is generated. Next, overall fitness of individuals in the population is evaluated. From the fitness values of individuals, more favorable ones are selected for forming the next generation. After selection operation, crossover between selected individuals is performed to produce the next generation. The ultimate step is to mutate individuals with a given probability for diversity of search traits. If the resulting generation has a solution which is optimal, then the algorithm is terminated. Otherwise, fitness evaluation, selection, crossover, and mutation operators are applied on the next generation iteratively.

Why the Genetic Algorithm?
As mentioned above, this algorithm is designed to replicate the natural selection process, i.e., survival of the fittest. It is usually implemented to solve various optimization problems. In our proposed implementation: cities are analogous to genes, string generated using cities is called a chromosome, while a fitness score equal to the path length of all the cities can be used to target a population.

Relevance of Parallel Execution:
The most time-consuming step in the algorithm is the fitness evaluation. If the population size is exceptionally large, fitness evaluation is very costly when done using sequential genetic algorithm. Another problem encountered in a sequential genetic algorithm is that it sometimes gets stuck in the zone of a local minima. To overcome the shortcomings mentioned above, Parallel Genetic Algorithm (PGA) is used.

Parallel Genetic Algorithms (PGA):
Implemented using CUDA to parallelize the genetic algorithm.

Genetic Algorithm:

1. Initialize the population randomly.
2. Determine the fitness of the chromosome.
3. Until done repeat:
   (a) Select parents.
   (b) Perform crossover and mutation.
   (c) Calculate the fitness of the new population.
   (d) Append it to the gene pool.

3. LITERATURE SURVEY – SIMILAR WORK

[1] Kang, S., Kim, S.S., Won, J., and Kang, Y.M., 2016. GPU-based parallel genetic approach to large-scale travelling salesman problem. *The Journal of Supercomputing*, *72*(11), pp.4399-4414 - It is difficult to efficiently solve TSP even with the large number of gene instances. This paper proposes an improved constructive crossover for TSP with which large number of genes can evolve by exploiting the parallel computing power of GPUs and an effective parallel approach to genetic TSP where crossover methods cannot be easily implemented in parallel fashion.

[2] Sena, G.A., Megherbi, D. and Isern, G., 2001. Implementation of a parallel genetic algorithm on a cluster of workstations: traveling salesman problem, a case study. *Future Generation Computer Systems*, *17*(4), pp.477-488 - The proposed algorithm is implemented using the Parallel Virtual Machine (PVM) library over a network of workstations. A master–slave paradigm is used to implement the parallel/distributed Genetic Algorithm (PDGA), which is based on a distributed-memory approach.

[3] Borovska, P., 2006, June. Solving the travelling salesman problem in parallel by genetic algorithm on multicomputer cluster. In *Int. Conf. on Computer Systems and Technologies* (pp. 1-6) - This paper investigates the efficiency of parallel computation of TSP using the genetic approach on a slack multicomputer cluster. For the parallel algorithm design functional decomposition of the parallel application has been made and the manager-workers paradigm is applied. Performance estimation and parallelism profiling have been made on the basis of MPI-based parallel program implementation.

[4] Harun Raşit Er, Dr. Nadia Erdoğan, March 2013, Parallel Genetic Algorithm to Solve Traveling Salesman Problem on MapReduce Framework using Hadoop Cluster, *Doi: 10.7321/jscse.v3.n3* - This paper shows a parallel genetic algorithm implementation on MapReduce framework in order to solve TSP. MapReduce is a framework used to support distributed computation on clusters of computers.

[5] Rajesh Matai, Surya Prakash Singh and Murari Lal Mittal (2010), Travelling Salesman Problem, An overview of Applications, Formulations, and Solution Approaches, Prof. Donald Devendra (Ed.), ISBN: 978-953-307-426-9, In Tech

4. METHODOLOGY

Genetic algorithm in terms of TSP

The concepts of genetic algorithm which have been extrapolated to solve TSP are explained below:

- Population: The problem statement consists of ‘n’ cities and we have to find the least cost path to traverse all cities exactly once and return to the start point. By permutation, we have n! total possible paths that can be generated. Hence, all the possible paths constituting these ‘n’ cities can be considered as the population, where the least cost path is the fittest species.
- Fitness Evaluation: To compare the fitness of any two paths in the population, we can directly compare the total cost of traversal. Hence, lower the cost, higher the fitness of the path.
- Tournament selection: A set of chromosomes are randomly selected and made to compete with one another so that the fittest chromosome can be utilized further. Hence, a specified number of paths are randomly selected from the population and compared in terms of cost. The least cost paths from here are selected and sent over for crossover and mutation to further optimize cost.
- Mutation: Mutation, when explained in biological terms, is the change in genetic sequence which causes diversity amongst organisms. In this case, mutation in a path is the swapping of randomly selected cities, to create an entirely different path.
- Crossover: When two parent chromosomes swap a part of their genetic sequence, it is termed as crossover. Similarly, we can swap specific cities from two parent paths and create an entirely new sequence.

Brief explanation of the functions from code with pseudo codes:

Selection

Chooses 2 parents for a particular tournament ID through tournamentSelection() and stores them in the parent_cities_d[] array in the global memory.

Pseudo Code:

```
selection () {
   tid <- blockDim.x * blockIdx.x + threadIdx.x;
   if (tid >= ISLANDS) return;

   parent1 <- tournamentSelection(tid);
   for (c=0; c<num_cities; c++)
      parent_cities_d[tid* (2*num_cities) +c] <- parent1[c];

   parent1 <- tournamentSelection(tid);
   for (c=0; c<num_cities; c++)
      parent_cities_d[tid* (2*num_cities) +num_cities +c] <- parent1[c];
}
```

TournamentSelection

This function declares the tournament array of the size of tournament multiplied by the number of cities to be covered in each. We iterate over the entire size of tournaments to subsamples a tournament from the existing population corresponding to the tournament is received as a parameter. Then it calls the function getFittestTourIndex() from the Header file to determine the fittest tournament index using which we return an array fittest_route containing the path lengths in the fittest route.

Mutation

Out of all the permutations present in the route connecting each city, we try to mutate or make changes. Randomly two cities are selected and swapped, causing a mutation. Here we take the offspring to check the modification undergone; if the required mutation rate is not achieved, we loop over and repeatedly mutate until we get the needed mutation rate.

Pseudo Code:

1) Current state mutation rate is calculated
2) If the mutation rate is lesser than the required value
3) Mutate again by swapping two randomly selected nodes
4)After all the mutation is performed, the final changes are stored

Crossover

This represents mating between individuals. Two individuals are selected using selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged thus creating a completely new individual (offspring).

Pseudo Code:

1) Crossover kernel: Perform merging of parents
population_d[tid*num_cities] = parent_cities_d[tid* (2*num_cities)];
parent_city_ptr[i]
2) tourarray[i] = population_d[tid*num_cities + i];
3) Chooses next valid city based on the last one in the route from each parent
4) Keep the better choice from both the parents by checking the one that is closer

L2Distance

Function for storing distance between two cities given their respective coordinates.

Pseudo Code:

1) From the passed coordinates of (x1, y1) of city1 and (x2, y2) of the connecting city
2) Distance is calculated

EvaluateRoute

Here the cost of all routes is added for traveling on that particular route and then normalized.
Distance, Population cost, and Population fitness are calculated.

Pseudo Code:

1) Distance connecting nodes is calculated
distance += citymap[population[i*num_cities + j]*num_cities + population[i*num_cities + j+1]];
2) Population Cost is directly proportional to distance
population_cost[i] = distance;
3) Population fitness is indirectly proportional to cost
population_fitness[i] = (1.0/population_cost[i]);

InitialiseRandomPopulation

Population Initialization is the first step in the Genetic Algorithm Process. The population is a subset of solutions in the current generation. Random Initialization is to Populate the initial population with completely unexpected solutions, allowing the entire range of possible solutions. Here Population P can also be defined as a set of chromosomes. The initial population P(0), which is the first generation, is created randomly.

Pseudo Code:

1) Randomly Population from the cities
2) This value is being stored after which the route is calculated

GetPopulationFitness /GetFittestScore

The fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score. It calculates the fittest Population that's best to undergo cross over. Calculates fitness depending on the route

Pseudo Code:

1) evaluateRoute() function is called
2) Accordingly, fitness is calculated

Main function

The array "citymap" is initialized which stores the distance between any two cities i and j. The arrays which will be storing the population, population size and cost are dynamically initialized. The number of islands, defined in the constants file and the total population size, calculated by islands*(number of cities) are printed. The array "citymap" is initialized in the subsequent lines by calling the L2Distance function repeatedly for every two cities.

A random population is initialized and the minimum cost of this initial population is calculated and displayed. Device variables are dynamically initialized using calloc. Before proceeding with the genetic algorithm, initial population fitness is calculated across all islands parallelly.

The code now proceeds to execute the parallel genetic algorithm for a particular number of generation (defined by the variable num_generations). Inside the for loop for each generation, first the selection process is carried out, followed by crossover, mutation and calculation of fitness parallelly across all blocks and threads. If the generation number is a multiple of 100, the iteration and minimum distance is printed. Finally, after num_generation iterations, the cost of the optimal path is printed, along with the total time taken.

Constants

Some constant values, which have been used throughout the code files are in the "constants.c" file and explained below:

- We defined the number of blocks to be 1 and the number of threads to be 200. Hence the grid size, defined by the variable "ISLANDS" has a value of 200, obtained by the multiplication of the blocks and threads.
- Num_generations: defines the number of iterations or the number of generations after which we terminate our program.
- Mutation_ratio: gives a measure of the difference in the paths before and after mutation.
- Print_interval: stores the number of iterations after which the cost is printed on the screen. We designated it a value of 100, hence the least cost is printed every 100 iterations.
- Num_cities: the number of cities, taken as 52.
- Tournament_size: given a value of 50. Hence, 50 paths are selected at random from the total number of paths and the costs are compared to find the least cost paths.
- City_x[] and city_y[]: 1D arrays which store the x and y coordinates of all the cities.

Later, we have varied the values of the above variables for analysing the output.

5. EVALUATION, RESULTS & ANALYSIS

```
Iteration:997200, min distance: 32205.419922
Iteration:997300, min distance: 29594.000000
Iteration:997400, min distance: 27995.566406
Iteration:997500, min distance: 30260.531250
Iteration:997600, min distance: 29576.947266
Iteration:997700, min distance: 31135.431641
Iteration:997800, min distance: 31147.582031
Iteration:997900, min distance: 32381.023438
Iteration:998000, min distance: 32838.093750
Iteration:998100, min distance: 31539.525391
Iteration:998200, min distance: 30183.449219
Iteration:998300, min distance: 27975.156250
Iteration:998400, min distance: 28356.875000
Iteration:998500, min distance: 30847.650391
Iteration:998600, min distance: 28667.169922
Iteration:998700, min distance: 26691.419922
Iteration:998800, min distance: 28424.144531
Iteration:998900, min distance: 30787.609375
Iteration:999000, min distance: 30696.015625
Iteration:999100, min distance: 31353.750000
Iteration:999200, min distance: 30674.375000
Iteration:999300, min distance: 30450.347656
Iteration:999400, min distance: 32412.423828
Iteration:999500, min distance: 32789.675781
Iteration:999600, min distance: 30826.119141
Iteration:999700, min distance: 31125.361328
Iteration:999800, min distance: 30743.935547
Iteration:999900, min distance: 29073.958984
time: 371.737030,  min distance: 29453.427734
```

Output of Sequential Genetic algorithm
(Did not converge till 1 million iterations)

Google Colab GPU allocated:

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.67       Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   62C    P8    10W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

1.

```
!./main

Num islands: 200
Population size: 10400
min distance: 22489.226562
Iteration:100, min distance: 11001.623047
Iteration:200, min distance: 10259.994141
Iteration:300, min distance: 9489.960938
Iteration:400, min distance: 9367.638672
Iteration:500, min distance: 8961.093750
Iteration:600, min distance: 8943.388672
Iteration:700, min distance: 8943.388672
Iteration:800, min distance: 8941.551758
Iteration:900, min distance: 8941.551758
Iteration:1000, min distance: 8941.551758
Iteration:1100, min distance: 8941.551758
Iteration:1200, min distance: 8935.810547
Iteration:1300, min distance: 8849.576172
Iteration:1400, min distance: 8849.576172
Iteration:1500, min distance: 8849.576172
Iteration:1600, min distance: 8849.576172
Iteration:1700, min distance: 8849.576172
Iteration:1800, min distance: 8838.540039
Iteration:1900, min distance: 8838.540039
time: 3.779845,  min distance: 8838.540039
```

const int num_blocks = 1;
const int num_threads = 200;
const int num_generations = 2000;
const float mutation_ratio= 0.05;

2.

```
!./main

Num islands: 400
Population size: 20800
min distance: 22489.226562
Iteration:100, min distance: 9332.084961
Iteration:200, min distance: 8806.058594
Iteration:300, min distance: 8702.508789
Iteration:400, min distance: 8702.508789
Iteration:500, min distance: 8655.489258
Iteration:600, min distance: 8631.796875
Iteration:700, min distance: 8631.796875
Iteration:800, min distance: 8631.796875
Iteration:900, min distance: 8631.796875
Iteration:1000, min distance: 8631.796875
Iteration:1100, min distance: 8631.796875
Iteration:1200, min distance: 8631.796875
Iteration:1300, min distance: 8631.796875
Iteration:1400, min distance: 8631.796875
Iteration:1500, min distance: 8631.796875
Iteration:1600, min distance: 8631.796875
Iteration:1700, min distance: 8631.796875
Iteration:1800, min distance: 8631.796875
Iteration:1900, min distance: 8631.796875
time: 6.480860,  min distance: 8631.796875
```

const int num_blocks = 1;
const int num_threads = 400;
const int num_generations = 2000;
const float mutation_ratio= 0.05;

3.

```
!./main

Num islands: 400
Population size: 20800
min distance: 22489.226562
Iteration:100, min distance: 9332.084961
Iteration:200, min distance: 8806.058594
Iteration:300, min distance: 8702.508789
Iteration:400, min distance: 8702.508789
Iteration:500, min distance: 8655.489258
Iteration:600, min distance: 8631.796875
Iteration:700, min distance: 8631.796875
Iteration:800, min distance: 8631.796875
Iteration:900, min distance: 8631.796875
Iteration:1000, min distance: 8631.796875
Iteration:1100, min distance: 8631.796875
Iteration:1200, min distance: 8631.796875
Iteration:1300, min distance: 8631.796875
Iteration:1400, min distance: 8631.796875
Iteration:1500, min distance: 8631.796875
Iteration:1600, min distance: 8631.796875
Iteration:1700, min distance: 8631.796875
Iteration:1800, min distance: 8631.796875
Iteration:1900, min distance: 8631.796875
time: 6.450880,  min distance: 8631.796875
```

const int num_blocks = 2;
const int num_threads = 200;
const int num_generations = 2000;
const float mutation_ratio= 0.05;

(Notice that the code saturated at same iteration)

4.

```
!./main

Num islands: 200
Population size: 10400
min distance: 22489.226562
Iteration:100, min distance: 10344.806641
Iteration:200, min distance: 10039.929688
Iteration:300, min distance: 9950.605469
Iteration:400, min distance: 9950.605469
Iteration:500, min distance: 9950.605469
Iteration:600, min distance: 9950.605469
Iteration:700, min distance: 9950.605469
Iteration:800, min distance: 9950.605469
Iteration:900, min distance: 9950.605469
Iteration:1000, min distance: 9950.605469
Iteration:1100, min distance: 9950.605469
Iteration:1200, min distance: 9950.605469
Iteration:1300, min distance: 9950.605469
Iteration:1400, min distance: 9950.605469
Iteration:1500, min distance: 9950.605469
Iteration:1600, min distance: 9950.605469
Iteration:1700, min distance: 9950.605469
Iteration:1800, min distance: 9950.605469
Iteration:1900, min distance: 9950.605469
time: 3.785914,  min distance: 9950.605469
```

const int num_blocks = 1;
const int num_threads = 200;
const int num_generations = 2000;

const float mutation_ratio= 0.10;

5.

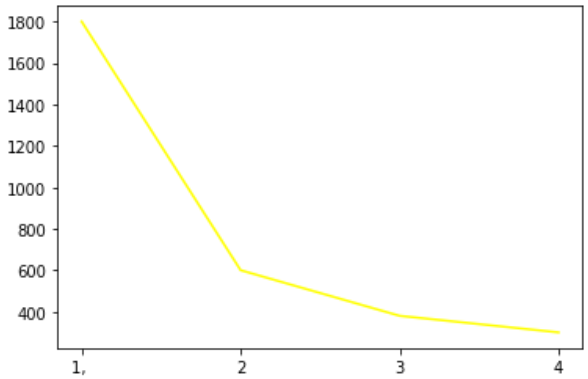
```
!./main

Num islands: 200
Population size: 10400
min distance: 22489.226562
Iteration:100, min distance: 11094.058594
Iteration:200, min distance: 9723.980469
Iteration:300, min distance: 9449.815430
Iteration:400, min distance: 9432.724609
Iteration:500, min distance: 9432.724609
Iteration:600, min distance: 9432.724609
Iteration:700, min distance: 9432.724609
Iteration:800, min distance: 9432.724609
Iteration:900, min distance: 9432.724609
Iteration:1000, min distance: 9432.724609
Iteration:1100, min distance: 9432.724609
Iteration:1200, min distance: 9432.724609
Iteration:1300, min distance: 9432.724609
Iteration:1400, min distance: 9432.724609
Iteration:1500, min distance: 9432.724609
Iteration:1600, min distance: 9432.724609
Iteration:1700, min distance: 9432.724609
Iteration:1800, min distance: 9432.724609
Iteration:1900, min distance: 9432.724609
time: 4.695923,  min distance: 9432.724609
```

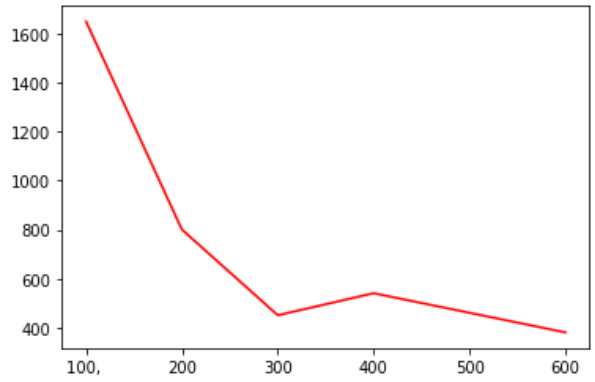
const int num_blocks = 1;
const int num_threads = 200;
const int num_generations = 2000;
const float mutation_ratio= 0.05;
tournament size=100

Graphical Representations varying Parameters:-

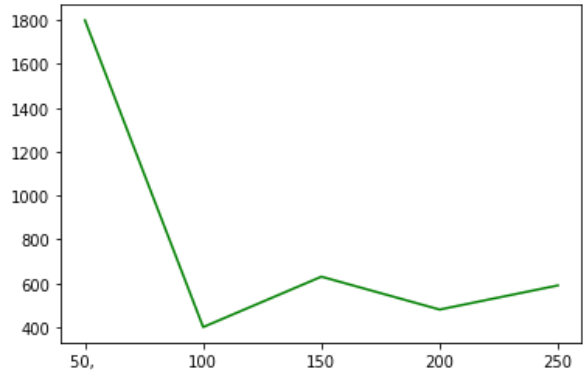
Varied Number of Blocks:



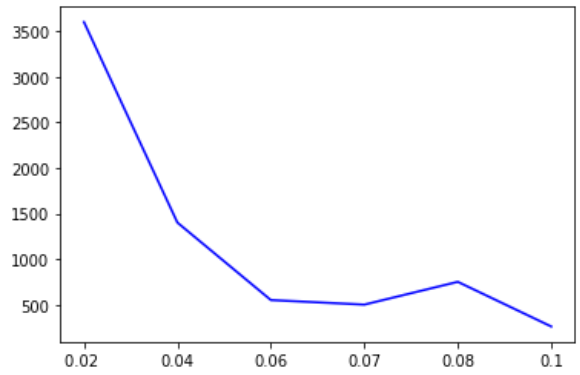
Varied Number of Threads:



Varied Tournament Sizes:



Varied Mutation Ratio:



6. CONTRIBUTIONS

Mutation and Route Evaluation: Sathvika B. Mahesh
Selection function using tournament selection criteria: Anshuman Sinha
Using CUDA to parallelize Genetic Algorithm: Saumya Karri

7. FURTHER IMPLEMENTATIONS

A lot of further improvements and implementations can be made to the project which are listed below:

- Usage of OpenMP to further parallelize the algorithm in a single processor.
- Add the option of a csv file import for the details of the cities and corresponding costs.
- Make a user-friendly interface and mount in on a server.

8. CONCLUSION

We used a parallelized genetic algorithm to successfully solve the Traveling Salesman Problem using CUDA on Google Colab's publicly available GPU. A sequential approach to the Genetic Algorithm turned out to be computationally unsuccessful since the execution did not converge even upon the completion of 1 million iterations. Using efficiently parallelizable optimization algorithms for solving equivalent TSPs of vehicle routing problems is critical in minimizing any intelligent transportation system costs.