GPU Project
Parallelizing Algorithms to solve Travelling Salesman Problem

Vishaal Udandarao (2016119)
Raghav Sood (2016259)

1st April 2019

0.1 Literature Survey

Travelling Salesman Problem(TSP) was first proposed in the 1930s as a mathematical problem. Given a set of cities and distance between every pair of cities, the travelling salesman problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. The first solution to TSP was proposed around the mid-1950s. It was shown that TSP is an NP-hard problem.

Existing work on TSP includes techniques used to speed up the otherwise slow algorithm. Focus has generally been on using 3 kinds of techniques to speed up the algorithm. a) Local Search based techniques , b) Population based techniques like Genetic Algorithms and c) Swarm Intelligence Techniques like Ant Colony Optimisation.

We plan to build upon these techniques and improve them by trying to parallelize them with the help of CUDA in this project.

0.2 Analysis of the Algorithm

TSP comes under the class of NP hard problems as the worst-case running time for any algorithm for the TSP might increase superpolynomially with the number of cities. Many sequential algorithms of varying time complexity have been proposed to solve the TSP. The naive sequential algorithm to solve TSP is to generate all possible permutations of the cities, calculate cost of every permutation and keep track of minimum cost permutation. This algorithm runs in O(n!) time. Another proposed algorithm makes use of the dynamic programming principles. It is the Held–Karp algorithm that solves the problem in time $O(n^2 2^n)$.

The algorithms that we are employing to solve the TSP are:

- Genetic Algorithm

- Ant Colony Optimization

  The available parallelisms and the task dependency graphs for these algorithms are explained below.

0.2.1 Genetic Algorithm

Genetic Algorithm is an optimisation algorithm based on the theory of Natural Selection proposed by Charles Darwin. A genetic algorithm consists of 5 stages:

Initialisation

Each solution is called a chromosome which is composed of multiple genes(Usually binary values). The entire set of solutions is a population. In our case a chromosome is a randomly chosen permutation of the N cities. A city on this chromosome becomes its gene

Fitness Function

Fitness is a score given to a solution (chromosome) of the problem. We take the total distance travelled to traverse the chromosome as its fitness score. Lesser the fitness score, fitter is the chromosome.

Selection

Now in each iteration two fittest chromosomes i.e. permutations with least overall distances,are chosen (selection) from the population.

Crossover

The two parents are used to produce the two offsprings. Thus is done by a merging/crossover of the two parents. So the ordering of cities in the offsprings is a combination of the 2 parents.

Mutation

Some genes of the offspring can randomly change their values to maintain diversity.

At the end of each iteration, we add the offsprings to the population, removing the 2 least fittest chromosomes. The last 3 steps are repeated iteratively till convergence,i.e we find the least distance path.

The obvious parallelism which exists here is, mapping activities of different chromosomes to different threads. Since all chromosomes are doing the same job, they finish nearly at the same time. This strategy can help us implement parallelism. Activities of different chromosomes have no task dependency thus it is easy to do mapping of tasks to processes.

0.2.2 Ant Colony Optimization

It is a method of generating "good solutions" to the TSP using a simulation of an ant colony called ACS (ant colony system). It models behaviour observed in real ants to find short paths between food sources and their nest, resulting from each ant's preference to follow trail pheromones deposited by other ants.

ACS sends out a large number of virtual ant agents to explore many possible routes on the graph. Each ant chooses the next city to visit with some probability drawn based on a heuristic that depends on the distance to the city and the amount of pheromone deposited on the edge to that city. The ants explore, deposit pheromone on each edge that they cross, and finally complete the tour. The ant which completed the shortest tour deposits virtual pheromone along its complete tour route.

There are two major steps in the solution of the TSP using ACO. they are:

Edge Selection

Each ant selects which edge it has to move along based on a heuristic that depends on the closeness of the next city to the current city as well as the amount of pheromone trail present on the edge. the probability with which the kth ant will move from city x to city y is:

$$p_{xy}^{k} = \frac{(\tau_{xy}^{\alpha})(\eta_{xy}^{\beta})}{\sum_{z \in \mathrm{allowed}_x} (\tau_{xz}^{\alpha})(\eta_{xz}^{\beta})}$$

Pheromone Update

After all the ants have completed their edge selection, the pheromone updated is performed.

$$\tau_{xy} \leftarrow (1-\rho)\tau_{xy} + \sum_k \Delta\tau_{xy}^k$$

The inherent parallelism that exists in the ant colony optimization algorithm to speed up the solution construction for TSP is to simultaneously simulate the tours of all ants. Thus, the available parallelism is to consider each ant as a separate thread and let each such virtual ant (thread) run its own tour simulation (edge selection and pheromone update). Thus, the tasks in this case will be the edge selection and pheromone update stages of the ACO algorithm.

0.3 Parallelization Strategy and Implementation

0.3.1 Genetic Algorithm

The initial generation of the entire genetic algorithm is time consuming. We plan to parallelize this portion as heavily as possible. Initialization of the entire population, i.e creating the random chromosomes is independent of one another. We can thus assign one thread for creating one chromosome.

Another parallelism which exists here is, calculating the fitness scores for the entire population. Sequentially it is done in O(n) time. This can be done completely parallely as there is no task dependency on each other.

In selection step we need to check which are the two fittest chromosomes. This is like the max operation. We can use a parallel reduce operation to parallelize this max operation.

Crossover requires to chromosomes to interact with one another. Since this means interaction between two threads, we instead create a local copy of the chromosomes on each other threads to reduce the task interaction between threads and increase parallelism.

Mutation of the offsprings are again independent of each other and can be processed in parallel.

Finally updating the population requires finding the least fittest chromosome. This is similar to the min operation and we again use parallel reduce operation to parallelize this min operation.

0.3.2 Ant Colony Optimization

The most computationally expensive part of the algorithm is the simulation of the ants for the tour construction and pheromone update stages. The computation required for selecting the next cities for each ant is highly parallelizable as this can be run across multiple blocks and threads. During the tour construction (edge selection) phase of the algorithm for each ant, the computation required to find the next city in the tour can be parallelized across all cities. Also, there is a high amount of data locality when accessing the data structure that stores distances between cities. This could mean that using shared memory to store these distance is a viable option to improve speedup.

The pheromone update stage of the algorithm can also be parallelized significantly. The pheromone values for each edge in the graph must be updated for every ant in the simulation. This depicts the need for synchronization between ants that want to update the same edge. This can be done either using atomic operations or by the use of barriers such as cudaDeviceSynchronize().

Our implementation for this stage of the project contains the tour

Table 1: Table showing results for GA

	bays29	att48
CPU time	721.32	1003.18
GPU time	34.1	45.2
Speedup	21.205	22.288

construction (edge selection) stage of the algorithm. We have done both the sequential and parallel implementations of the same and have shown our results. The hyperparameters that we have selected for the algorithm are: alpha=1, beta=2, rho=0.5, max-iterations=30 and Q=100. We have tested our algorithm on two distinct datasets that contain 48 and 29 cities respectively. With regards to the kernel statistics, we have used a grid size of N/32, where N is the number of cities in the dataset and a block size of 32.

0.4 Results

The following results have been generated on two test sets, one containing N=29 cities and the other N=48 cities.

0.4.1 Genetic Algorithm

We ran the complete sequential version of Genetic Algorithm on the two test sets. Also currently we have parallelized the initialization, selection and crossover steps of genetic algorithm. As we mentioned above the major improvements which we expect to get will be covered in these parts itself. The remaining steps mutation and the post mutation have still not been parallelized but the improvements from this part will not be as significant as what we see now. Trace 1 is CPU time. Trace 0 is GPU time. Trace 2 i speedup

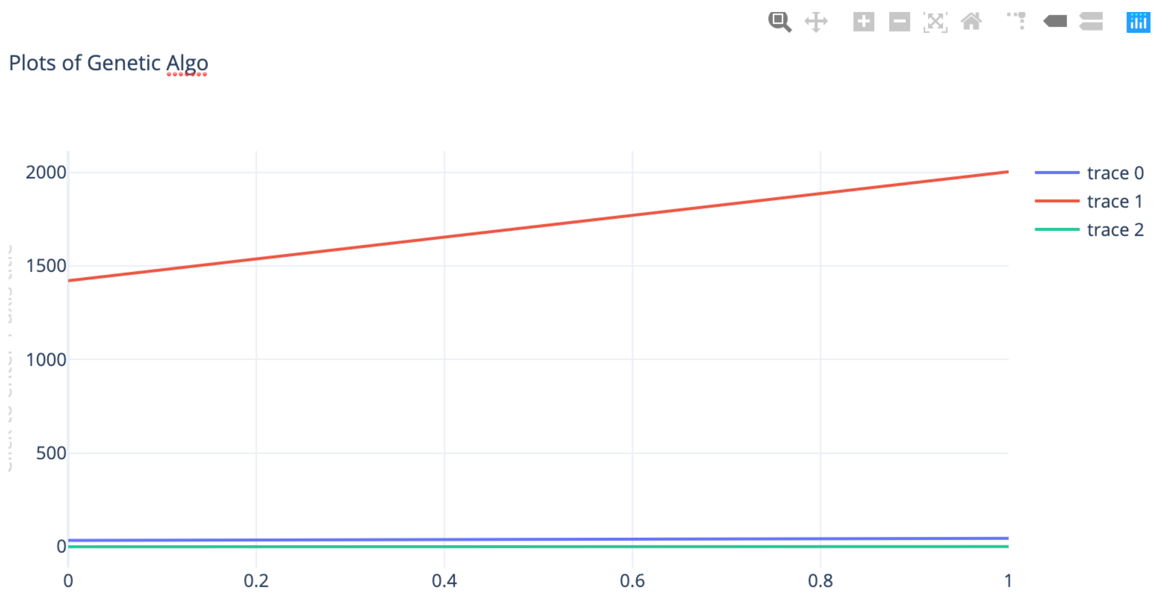

Figure 1: Plot showing the CPU times, GPU times and speedups obtained on the two datasets

Table 2: Table showing results for ACO

	bays29	att48
GPU time	53.22	139.16
CPU time	611.29	1605.6
Speedup	11.486	11.538

0.4.2 Ant Colony Optimization

We generated the following results on the two datasets.

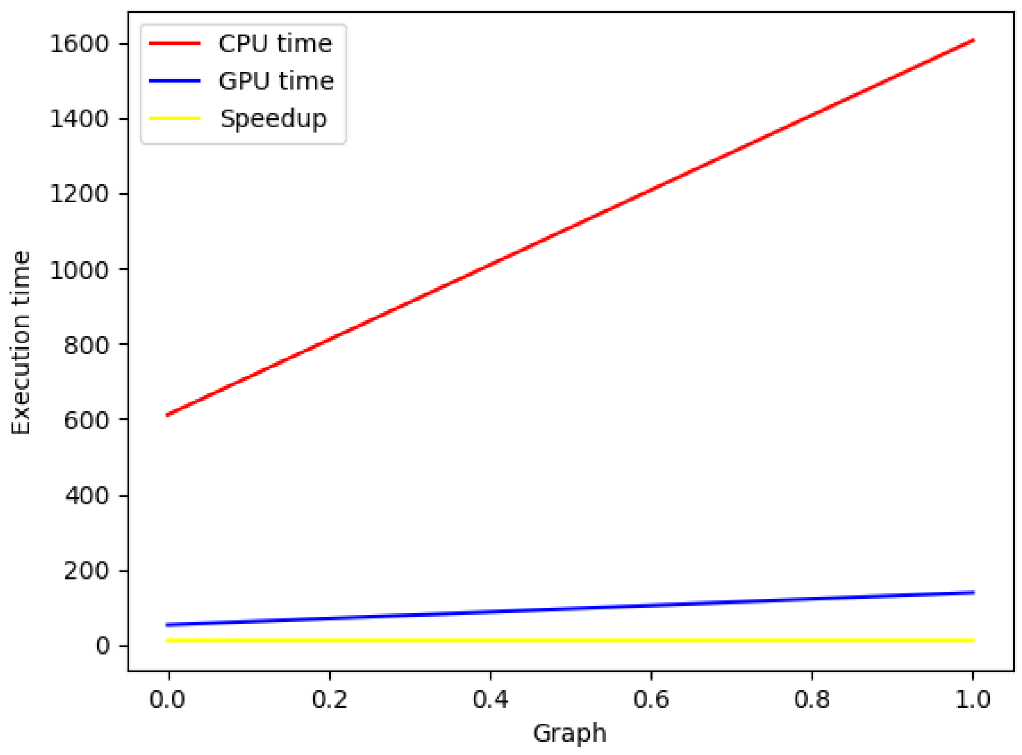

Figure 2: Plot showing the CPU times, GPU times and speedups obtained on the two datasets

0.5 Final Deliverables

0.5.1 Genetic Algorithm

Our final deliverables for the Genetic algorithm will be to parallelize the mutation and post-mutation steps of the Genetic Algorithm and to analyse the speedups obtained by the genetic algorithm.

0.5.2 Ant Colony Optimization

Our final deliverables for the ACO algorithm will be to complete the basic ACO algorithm with considerable speedup in the parallel algorithm. We will also be implementing the Max-Min, Elitist and Rank Based (if time permits) versions of the ACO algorithms. We will then analyse and compare the speedups obtained between the various versions of the ACO algorithm.

0.6 References

1. http://www.scholarpedia.org/article/Ant$_{c}olony_{o}ptimizationMAX-$
$MIN_{a}nt_{s}ystem2.https : //www.researchgate.net/publication/225111688_{C}UDA-$
$Based_{G}enetic_{A}lgorithm_{o}n_{T}raveling_{S}alesman_{P}roblem3.https : //ieeexplore.ieee.org$