Parallelizing the Traveling Salesman Problem Using Ant Colony Optimization on a GPU

Aayush Poudel, Matt Nappo
{apoudel, mnappo}@u.rochester.edu

CSC 258, Prof. Chen Ding
University of Rochester
Spring 2023

***Abstract*— This paper presents a GPU implementation of the Traveling Salesman Problem (TSP) using CUDA. We compare the performance of our GPU implementation against a CPU implementation and an OpenMP implementation on the CPU. We compare the running times of both implementations, as we've verified the correctness of the algorithms. Our experiments show that the GPU implementation is significantly faster than the CPU and OpenMP implementations. We also validate the optimality of our solution against the linear programming based Concorde TSP solver [9]. Overall, our results strongly show that a GPU implementation of the TSP can provide significant speedup over CPU implementations, making it a promising approach for solving large-scale TSP instances.**

I. Introduction

The Traveling Salesman Problem (TSP) is a well-known optimization problem in computer science and operations research. It involves finding the shortest possible route that a salesman can take to visit a given set of cities and return to the starting city, visiting each city exactly once. The TSP has been shown to be NP-hard, which means that finding the exact solution to this problem for large instances is computationally intractable. As a result, there has been a lot of interest in developing efficient approximation algorithms for the TSP.

In this paper, we present an implementation of the TSP using both CPU and GPU-based algorithms. We also compare the runtimes of these algorithms and evaluate their performance. We provide a baseline, fully-sequential CPU implementation, an OpenMP implementation on the CPU, and a GPU implementation using C++/CUDA.

This paper is organized as follows: in Section 2, we provide a brief overview of the TSP and its relevance in operations research. In Section 3, we describe the methodology used in our implementation. In Section 4, we present our results and evaluate the performance of our algorithms. Finally, in Section 5, we conclude the paper and discuss possible future work.

A. Methodology

We use Ant Colony Optimization (ACO) to solve the Traveling Salesman Problem. ACO is a metaheuristic algorithm inspired by the foraging behavior of ants, which has been shown to be effective in solving combinatorial optimization problems [1].

Our approach involves constructing a graph where the cities represent the nodes and the edges represent the distances between them. The ants then traverse this graph, depositing pheromones along the edges they travel. The amount of pheromones deposited is proportional to the quality of the solution obtained by the ant. The pheromone trail serves as a form of communication between the ants, allowing them to collectively explore the search space and converge toward an optimal solution.

We implemented ACO using C++ and CUDA. We also developed a CPU version of the algorithm and a variant that uses OpenMP for parallelization on the CPU. We compared the performance of these implementations in terms of runtime on the same hardware.

To evaluate the quality of our solutions, we used a standard benchmark of TSP instances from various countries, obtained from the University of Waterloo [6]. We used the runtime as the primary metric for comparing the performance of our different implementations. We also checked the optimality of our solutions against the Concorde TSP solver [9].

II. Implementation

A. Sequential CPU Implementation

The pheromone update rule is a crucial component of the ant colony optimization algorithm. It allows ants to learn from the paths taken by their peers, and over time, converge to a near-optimal solution. The update rule is based on the principle that ants deposit pheromones on the edges they traverse, and these pheromones attract other ants to follow the same path. In this section, we describe the pheromone update rule used in our implementation of the ant colony optimization algorithm for solving the TSP. We provide the mathematical equations used to calculate the pheromone deposit and evaporation rates, and explain how these rates affect the convergence and exploration properties of the algorithm [1].

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij} + \Delta\tau_{ij}$$

where τ_{ij} is the pheromone level on edge (i,j), ρ is the pheromone decay rate, and $\Delta\tau_{ij}$ is the amount of pheromone deposited on edge (i,j) by the ants. The amount of pheromone deposited is calculated as:

$$\Delta\tau_{ij} = \frac{1}{L_k}$$

where L_k is the length of the tour constructed by ant k. The attractiveness parameter that the ants use to select the next edge in each tour construction is given by the following equation.

$$\eta_{i,j} = \frac{1}{d_{i,j}} \qquad \tau_{i,j}^{(t)} \leftarrow (1-\rho)\tau_{i,j}^{(t-1)} + \sum_{k=1}^{m} \Delta\tau_{i,j}^{(k)}$$

where $d_{i,j}$ is the distance between cities i and j, and ρ is the pheromone decay coefficient. The update equation for the pheromone trail is given by $\tau_{i,j}^{(t)}$, where t denotes the iteration number, and k denotes the ant. Given these details, the attractiveness matrix that is used by the ants is defined as follows.

$$a_{ij} = \begin{cases} \frac{[\tau_{ij}(t)]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{k\in\mathscr{J}_i}[\tau_{ik}(t)]^{\alpha}[\eta_{ik}]^{\beta}}, & \text{if } j \in \mathscr{J}_i \\ 0, & \text{otherwise} \end{cases}$$

where $\mathscr{J}_i$ is the set of cities that are adjacent to city i, α and β are parameters that control the relative importance of the pheromone trail and the heuristic information, and η_{ij} is the inverse of the distance between cities i and j:

B. OpenMP Implementation

The OpenMP is almost identical to the serial CPU implementation. Since ants within an iteration are largely independent, the sequential for-loop iterating over the ants can be massively parallelized. We achieved this with the

```
#pragma omp parallel for
```

compiler directive. For large graphs, it is unlikely that the same edge in τ will be updated by different ants at the same time. Thus, using atomics or a lock is a fast and correct solution to prevent a data race on τ since it will introduce very little serialization [4]. We configured OpenMP to use 16 threads, and saw a great speedup with no loss in accuracy. We will further discuss program evaluation in the next section.

C. GPU Implementation

A critical decision in designing a GPU algorithm is to define the granularity of parallelism. The algorithm should clearly define what tasks each grid, block, and thread are responsible for, and how the data will be shared and synchronized. Our algorithm is based on the *coarse-grained* approach outlined in [4], where each thread within a block acts as an ant. That is, each thread constructs a tour by sequentially performing the necessary probability calculations [1]:

$$P(i,j) = \frac{A(i,j)}{\sum_{j'\in\mathscr{J}_i} A(i,j')}$$

Algorithm 1: Ant Colony Optimization for the TSP

```
Input : Adjacency matrix $adjacency\_matrix$ of
        size $num\_nodes \times num\_nodes$
        representing the TSP problem
Input : Number of ants $num\_ants$
Input : Maximum number of iterations $max\_iters$
Input : Evaporation rate $\rho$
Input : Initial pheromone level $\tau_0$
Input : Heuristic information $A$ of size
        $num\_nodes \times num\_nodes$
Output: Hamiltonian cycle of minimum length
1: Initialize $\tau$ to $\tau_0$ ; for $i \leftarrow 1$ to $max\_iters$ do
2:     Create $num\_ants$ ants, each starting at a
         random node ; for $j \leftarrow 1$ to $num\_ants$ do
3:         Run ant $j$, which will update $\tau$ and keep
             track of the best solution found so far ;
4:     Evaporate pheromone trails by multiplying $\tau$ by
         $(1-\rho)$ ; Update pheromone trails with new
         pheromone by adding $\Delta\tau$ ;
5: return Best Hamiltonian cycle found
6: Function $run\_ant(adjacency\_matrix,$
     $num\_nodes, \tau, A, iter)$:
7:     Initialize ant's path with node 0 ; Initialize
         boolean array $visited$ of length $num\_nodes$
         to all false ; while path is not complete do
8:         Get unvisited neighbors of the last node
             visited ; Calculate attractiveness of
             unvisited neighbors using heuristic
             information $A$ ; Sample probability
             distribution to choose next node to visit ;
             Add next node to path and mark as visited
             ; Update pheromone trail on edge just
             traversed ;
9:     If this ant's path is better than previous paths,
         update $iter$ with the new path ;
```

Since the τ matrix is updated after each iteration, full device synchronization occurs after all the ants construct a tour. This thread barrier ensures all ants finish their tour construction before modifying any overlapping shared memory. τ is then updated in parallel, minimizing over the ants' tours to converge toward an optimal path.

We implemented the tour construction process as one kernel, which runs with $m/32$ blocks and 32 threads per block. The τ update is another kernel, which runs after tour construction.

III. Evaluation

We tested both models on the cycle computers with the following specifications for the CPU: x86 64-bit Intel Xeon Gold 5218 2.3GHz. For the GPU: NVIDIA GTX 1080.

The parameters used during all evaluation are given below.

$$k = 100 \quad \text{(number of iterations)}$$
$$m = \{512, 1024, 2048\} \quad \text{(colony size)}$$
$$\alpha = 0.5 \quad (\tau\text{-weight})$$
$$\beta = 0.2 \quad (\eta\text{-weight})$$
$$\rho = 0.7 \quad (\tau\text{-decay rate})$$
$$t = 16 \quad \text{(number of OpenMP threads)}$$

We evaluated our implementations on the following datasets:

Dataset	Nodes (N)	Source
`ts11.tsp`	11	(Ours)
`dj38.tsp`	38	[6]
`qa194.tsp`	194	[6]

This selection of datasets provided ample variance in problem sizes without requiring extreme execution time. We ran three trials for each configuration and took the average. Full data is available in the appendix.

A. Data Collected

The following charts show the runtimes in seconds of each of our implementations across all permutations of colony sizes and datasets. Each bar is the average runtime of the tree trials.

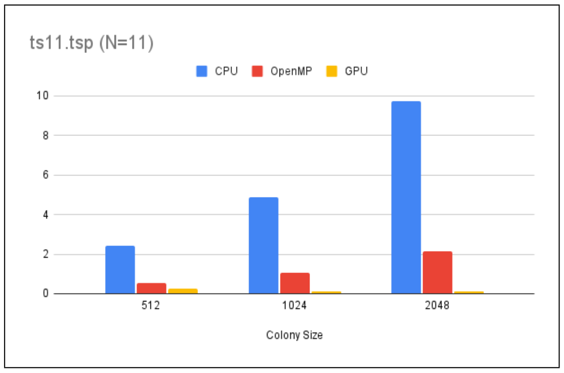

Fig. 1. `ts11.tsp` dataset runtimes

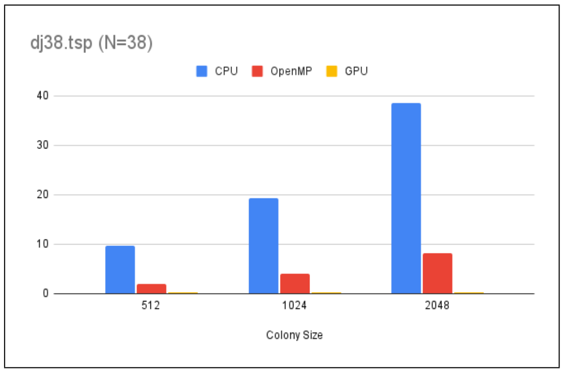

Fig. 2. `dj38.tsp` dataset runtimes

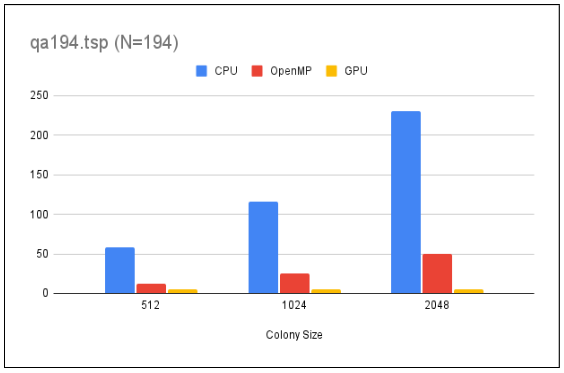

Fig. 3. `qa194.tsp` dataset runtimes

We see an incredible parallel speedup. In fact, the speedup grows with the problem size. Below is a table of GPU speedup, calculated by diving the CPU runtime by the GPU runtime.

GPU Speedup	N=11	N=38	N=194
512 ants	9.7x	35.9x	11.9x
1024 ants	52.5x	71.2x	23.5x
2048 ants	98.9x	139.2x	46.7x

Observe that as the colony size increases, so does the GPU speedup. This is because as the number of ants increases, the CPU implementation becomes increasingly sequential, while the GPU implementation becomes increasingly parallel. This observation is presented nicely in the below graph.

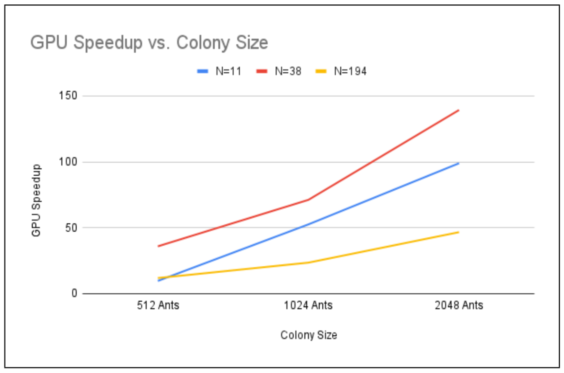

Fig. 4. GPU Speedup vs Colony Size

B. Error Discussion

The solutions generated by our implementation are often not exactly optimal. This is because larger problem sizes require more iterations, and thus more time, to converge. However, our implementation, when given more time to run, does converge toward an optimal solution, although the rate of convergence slows as the problem size increases. The error of our implementations may not be acceptable as a real TSP solver, especially since the error grows as the problem size increases. However, what is important for the scope of this project is that the error across our implementations does not vary. For

Error Variance	**512**	**1024**	**2048**
ts11	0.01	0.05	0.00
dj38	0.01	0.01	0.04
qa194	1.31	1.23	0.94

Fig. 5. Variance in Error Across Implementations and Configurations

example, if the CPU implementation has 20% error for a given configuration, the GPU implementation will have roughly 20% error under the same configuration (dataset, parameters, number of iterations, etc…). This can be seen in the above table, whose entries are the variances of error of the CPU, GPU, and OpenMP implementations of the given configuration. This is an average variance of 0.40, which is very low. Thus, our parallelized algorithms are correct with respect to the baseline, meaning that our parallel code is doing the same amount of work in a shorter amount of time. We measured error by comparing our path length to the path length of the linear programming-based solver Concorde, from University of Waterloo [9].

IV. Future Work

To further demonstrate the scalability of our implementation, we will evaluate on larger datasets for a greater number of iterations. We did not presently do this, as the CPU execution takes a long time. For example, running a colony of 2048 ants for 650 iterations on a dataset with 980 nodes took over 3 hours on the sequential CPU.

There are other possible optimizations that can be introduced to improve the core algorithm such as the 2-opt or 3-opt optimizations, and flying ant optimization as explored in [7]. The existing or extra hyperparameters can be tuned using a variety of parameter tuning heuristics as explored in [8]. Additionally, implementing reduction on the constructed tours after device synchronization will reduce the complexity of finding the optimal tour from $O(n)$ to $O(\log n)$.

V. Work Distribution

We managed to divide tasks fairly and efficiently.
Aayush primarily worked on

- Core graph library
- OpenMP Implementation
- CUDA *tour construction* kernel

Matt primarily worked on

- Sequential CPU implementation
- CUDA *pheramone update* kernel
- Benchmarking scripts and data collection

Although these were our primary tasks, we did spend a lot of time working together on the same pieces of critical code. All of the architecture and design decisions were made as a team, and we frequently employed pair programming strategies to optimize our workflow. We both spent equal time debugging and writing the paper.

Appendix

The appendix on the next page shows the full data we collected. Our complete source code can be viewed at `https://github.com/mattnappo/aco-tsp`

References

[1] Kochnderfer, Mykel J., and Tim A. Wheeler. "Ch. 19.6: Ant Colony Optimization." Algorithms for Optimization, The MIT Press, Cambridge, Massachusetts Etc., 2019.

[2] E. Duman and I. Or, "Precedence constrained TSP arising in printed circuit board assembly," Int. J. Prod. Res., vol. 42, no. 1, pp. 67–78, 2004.

[3] M. Dorigo and L. M. Gambardella, "Ant colonies for the travelling salesman problem," Biosystems., vol. 43, no. 2, pp. 73–81, 1997.

[4] B. A. M. Menezes, H. Kuchen, H. A. Amorim Neto and F. B. de Lima Neto, "Parallelization Strategies for GPU-Based Ant Colony Optimization Solving the Traveling Salesman Problem," 2019 IEEE Congress on Evolutionary Computation (CEC), Wellington, New Zealand, 2019, pp. 3094-3101, doi: 10.1109/CEC.2019.8790073.

[5] N. Christofides, "Bounds for the travelling-salesman problem," Oper. Res., vol. 20, no. 5, pp. 1044–1056, 1972.

[6] "National Traveling Salesman Problems," Uwaterloo.ca. [Online]. Available: `https://www.math.uwaterloo.ca/tsp/world/countries.html`.

[7] F. Dahan, K. El Hindi, H. Mathkour, and H. AlSalman, "Dynamic flying ant colony optimization (DFACO) for solving the traveling salesman problem," Sensors (Basel), vol. 19, no. 8, p. 1837, 2019.

[8] A. F. Tuani, E. Keedwell, and M. Collett, "Heterogenous Adaptive Ant Colony Optimization with 3-opt local search for the Travelling Salesman Problem," Appl. Soft Comput., vol. 97, no. 106720, p. 106720, 2020.

[9] William Cook, et. al., "Concorde TSP Solver," University of Waterloo, 2003. Version 03.12.19. Available: `https://www.math.uwaterloo.ca/tsp/concorde.html`.

TABLE I

CPU Runtime

CPU	**512**	**1024**	**2048**
ts11	2.428164	4.880486	9.724462
	2.427676	4.877515	9.765008
	2.425684	4.874685	9.767725
avg	**2.427174667**	**4.877562**	**9.752398333**
dj38	9.666022	19.287058	38.541475
	9.637466	19.27888	38.537591
	9.634692	19.29701	38.580878
avg	**9.64606**	**19.28764933**	**38.55331467**
qa194	57.83523	115.618977	231.17143
	57.85084	115.628404	230.379893
	57.835023	115.68279	230.460977
avg	**57.84036433**	**115.6433903**	**230.6707667**

TABLE II

OpenMP Runtime

OpenMP	**512**	**1024**	**2048**
ts11	0.551763	1.066565	2.130954
	0.548386	1.068954	2.120432
	0.552852	1.070685	2.144535
avg	**0.5510003333**	**1.068734667**	**2.131973667**
dj38	1.961675	4.048109	7.984701
	1.998093	3.99076	8.400066
	1.958204	4.090089	8.049795
avg	**1.972657333**	**4.042986**	**8.144854**
qa194	12.497604	25.0177	49.682099
	12.470494	25.022314	49.543378
	12.498432	25.091626	49.630387
avg	**12.48884333**	**25.04388**	**49.61862133**

TABLE III

GPU Runtime

GPU	**512**	**1024**	**2048**
ts11	0.538082	0.093802	0.095235
	0.106682	0.093719	0.092624
	0.107224	0.090951	0.107994
avg	**0.2506626667**	**0.092824**	**0.09861766667**
dj38	0.279474	0.278424	0.277911
	0.270265	0.27521	0.276967
	0.256206	0.259581	0.275723
avg	**0.2686483333**	**0.2710716667**	**0.276867**
qa194	4.863215	4.885787	4.950761
	4.87723	4.923453	4.93677
	4.879254	4.925176	4.946234
avg	**4.873233**	**4.911472**	**4.944588333**