GPU Ant Colony Optimisation for Traveling Salesman Problem CUDA Assignment

Maria Wysoglad

April 28, 2025

1 Introduction

The Traveling Salesman Problem (TSP) is a classical NP-hard optimization problem where a salesman must visit each city exactly once and return to the starting point, minimizing the total travel distance. Even small instances of TSP can be computationally challenging, making it an excellent target for parallel and GPU-based computation.

One popular metaheuristic to solve TSP is Ant Colony Optimization (ACO). It simulates a swarm of artificial ants that explore paths through the graph using both random exploration and accumulated pheromone trails to reinforce better solutions. Over many iterations, ants collectively converge toward near-optimal solutions.

In this assignment, two different GPU-based ACO variants (QUEEN and WORKER) are implemented and compared to analyze their performance.

My solution was developed based on the following paper:

- J.M. Cecilia, J.M. García, *Enhancing Data Parallelism for Ant Colony Optimisation on GPUs*.

2 Explanation of both implementations

2.1 Worker Ant Version

Each GPU thread corresponds to one ant, constructing a tour independently:

- **Thread Layout**: One thread per ant. There are at most 256 threads in order to prevent memory problems. The number of blocks is chosen depending on the number of cities.
- **Synchronization**: Only global synchronization after each iteration.
- **Tour Building**: Each thread maintains its own tour and calculates distances.

2.2 Queen Ant Version

Each block builds one tour cooperatively:

- **Thread Layout**: One block per queen, one thread per city (or worker ant).
- **Synchronization**: Block-wide synchronizations after phases (setting tabu list, collecting probabilites or choosing the next city), global syncronization after iteration.
- **Tour Building**: Queen thread constructs the tour for entire block based on probabilities collected by threads and roulette mechanism.

3 Optimizations Applied

- Implemented scatter-gather algorithm for pheromone updates as described in paper.
- Optimized visited array stored in dynamically allocated shared memory, stored as bits to save memory.
- Warp-reduction for collecting probabilities in QUEEN.
- Storing data in local variables where possible instead of accessing memory multiple times.

4 Results and Performance Measurements

4.1 Setup

All experiments were run on the datasets d198, a280, lin318, pcb442, rat783, pr1002 that might be found in tsplib github repository The parameters used are:

- `NUM_ITER` = 1000
- `ALPHA` = 1
- `BETA` = 2
- `EVAPORATE` = 0.5
- `SEED` = [32, 42, 64]

  I run 3 experiments using different seeds in order to calculate average measurements and tours.
  REMARK: Due to Entropy GPU limitations, 3 experiments were run only on first 3 smaller datasets because running more than one experiment at once was problematic - I was disconnected from the environment. So, I run only 1 experiment with seed 42 on greater datasets and I skip standard deviation, moreover on greater datasets I run only 100 iterations due to Entropy time limits and the deadline.

4.2 Results table

Table 1: Best tour length

Implementation	d198	a280	lin318	pcb442
WORKER GRAPH	17908 ± 377	3301 ± 71	49975 ± 527	65034
WORKER	17908 ± 377	3301 ± 71	49975 ± 527	65034
QUEEN GRAPH	18078 ± 177	3299 ± 57	49860 ± 977	63237
QUEEN	18078 ± 177	3299 ± 57	49860 ± 977	63237

This table shows average best tour lengths for various datasets with standard deviation. The results are rounded to the nearest whole number.

Table 2: Best tour length on greater datasets

Implementation	rat783	pr1002
WORKER GRAPH	11355	336100
WORKER	11355	336100
QUEEN GRAPH	11540	344088
QUEEN	11540	344088

This table shows average best tour lengths for bigger datasets, it is separated because it contains results after 100 iterations due to Entropy time limit and deadline.

4.3 Performance Comparison Table

Table 3: Graph/Kernels execution time in ms

Implementation	d198	a280	lin318	pcb442
WORKER GRAPH	20	49	84	145
WORKER	20	49	85	147
QUEEN GRAPH	8	15	53	82
QUEEN	8	15	54	82

This table shows average execution time rounded to the nearest whole number. I skipped standard deviations because they were very little, near 0.

Table 4: Graph/Kernels execution time in ms for greater datase

Implementation	rat783	pr1002
WORKER GRAPH	1181	2843
WORKER	1181	2843
QUEEN GRAPH	916	2333
QUEEN	917	2334

Table 5: Total execution time in seconds

Implementation	d198	a280	lin318	pcb442
WORKER GRAPH	19.81 ± 0.01	48.92 ± 0.04	84.36 ± 1.35	145.31
WORKER	19.79 ± 0.03	48.91 ± 0.01	84.93 ± 1.03	146.72
QUEEN GRAPH	7.64 ± 0.01	14.87 ± 0.01	53.33 ± 1.48	82.23
QUEEN	7.66 ± 0.02	14.84 ± 0.00	53.89 ± 0.69	82.46

This table shows average with standard deviation calculated over all experiments.

Table 6: Total execution time in seconds (after 100 iterations)

Implementation	rat783	pr1002
WORKER GRAPH	119.01	286.75
WORKER	119.02	286.79
QUEEN GRAPH	91.66	233.38
QUEEN	91.71	233.46

Table 7: Comparison of my mean best tours to optimal results for WORKER

Dataset	My Tour	Optimal Result	Proportion
d198	17908	15780	1.1349
a280	3301	2579	1.2800
lin318	49975	42029	1.1891
pcb442	65034	50778	1.2808
rat783	11355	8806	1.2895
pr1002	336100	259045	1.2975

Table 8: Comparison of my mean best tours to optimal results for QUEEN

Dataset	My Tour	Optimal Result	Proportion
d198	18078	15780	1.1456
a280	3299	2579	1.2792
lin318	49860	42029	1.1863
pcb442	63237	50778	1.2454
rat783	11540	8806	1.3105
pr1002	344088	259045	1.3283

We can see that proportion after 100 iteration is worse, but it is still within the limit described in FAQ.

Table 9: Execution Kernel Time Comparison: Scatter-Gather vs Normal Algorithm WORKER

Implementation	d198 (ms)	a280 (ms)	lin318 (ms)
Scatter-Gather	20	49	84
Normal Algorithm	17	41	52
Difference (Scatter - Normal)	3	8	32

Table 10: Execution Kernel Time Comparison: Scatter-Gather vs Normal Algorithm QUEEN

Implementation	d198 (ms)	a280 (ms)	lin318 (ms)
Scatter-Gather	8	15	53
Normal Algorithm	2	3	6
Difference (Scatter - Normal)	6	12	47

We can see that scatter-gather optimization is actually quite inefficient in terms of performance, but I left this implementation to gain points, as scatter-gather was one of possible optimizations.

You can find the kernels I used in normal solution, as well as commented code running it in my code.

5 CUDA Features Outside Lecture Scope

- **CUDA Graphs**: Reduces kernel launch overhead by recording and replaying sequences of GPU operations.
- **curand Library**: Efficient parallel random number generation directly on the GPU.

6 Conclusion

Queen Ant works faster overall. Which implementation results in a better path depends on the dataset and the seed.

Both Worker Ant and Queen Ant implementations quite successfully solve the Traveling Salesman Problem using ACO on the GPU. The Queen Ant version benefits more from cooperation between threads at the block level, especially for larger instances. CUDA Graphs further optimize the execution time by minimizing kernel launch latency.

While working on my solution and writing the report, I used GitHub Copilot and ChatGPT.

References

- J.M. Cecilia, J.M. García, *Enhancing Data Parallelism for Ant Colony Optimisation on GPUs*, Journal of Parallel and Distributed Computing, 2012. DOI: 10.1016/j.jpdc.2012.01.002