Team #9

Project Document

Provided By:

Name	Sec	BN	ID
Ahmed Sayed Sayed Madbouly	1	3	9202111
Ahmed Hany Farouk	1	10	9202213

Provided to:

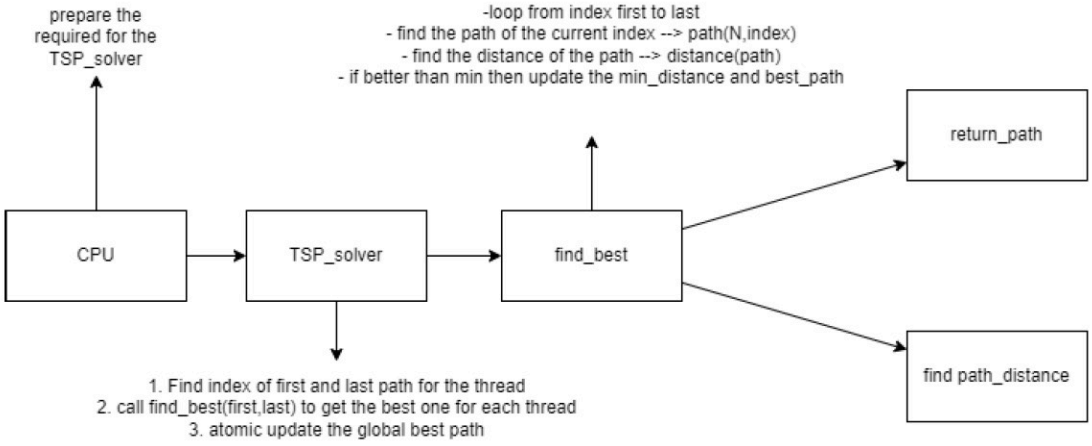
Dr/Dina Tantawy

Eng/Mohamed Abdellah

Problem Definition:

The Traveling Salesman Problem (TSP) is a classic optimization problem where a salesman needs to visit a set of cities exactly once and return to the starting city, covering the shortest possible route. Our project aims to parallelize the TSP solution using CUDA, leveraging the power of GPUs for efficient computation.

Final system design



1- In CPU we loaded the edges list and built the graph, calc the factorials from factorial[0] to factorial[N], launched the kennel on blocksize 1024 thread, and tried different numbers of blocks “We chose 15 blocks as big as enough”

2- each thread will start to calculate how many permutations it will need to work, then calculate the start and end index of these permutations.

3- for each permutation send the number of nods and the index of the permutation to another device function to calculate the permutation itself.

4-calculate the distance of the path in this single permutation.
5-for each thread find the best one in its set of paths.
6- each thread will store its best result in a global array with its index
7- only one thread ‘idx=0’ will iterate over this array and find the global best array, this step is to simulate the semaphore because cuda doesn't have a direct semaphore

Experiments and results

size	CPU time ‘s’	GPU time ‘s’
5	0.105746269226	0.7452752590179443
6	0.10415220260620117	0.5065548419952393
7	0.10396766662597656	0.5060691833496094
8	0.10614156723022461	0.5070180892944336
9	0.20439553260803223	0.5078833103179932
10	1.6080279350280762	0.6154828071594238
11	21.782139539718628	0.9082918167114258
12	258.97381711006165	4.420988321304321
13	4363.268090963364	57.912707805633545

Performance analysis

CPU Benchmarks (Theoretical)

The CPU implementation for solving the Traveling Salesman Problem (TSP) involves loading the edges list, building the graph, calculating factorials, and sequentially processing permutations. The theoretical benchmarks for the CPU are based on the

factorial growth rate of permutations, making the problem computationally intensive as the number of cities increases. The complexity is O(n!), where n is the number of cities.

GPU Benchmarks (Based on Implementation)

The GPU implementation leverages CUDA to parallelize the computation. Each thread calculates a subset of permutations and evaluates the path distance, with a lock mechanism to update the globally optimal path. The benchmarks for GPU performance were experimentally measured:

size	CPU time ‘s’	GPU time ‘s’
5	0.105746269226	0.7452752590179443
6	0.10415220260620117	0.5065548419952393
7	0.10396766662597656	0.5060691833496094
8	0.10614156723022461	0.5070180892944336
9	0.20439553260803223	0.5078833103179932
10	1.6080279350280762	0.6154828071594238
11	21.782139539718628	0.9082918167114258
12	258.97381711006165	4.420988321304321
13	4363.268090963364	75.342

Speedup of the GPU over the CPU

The speedup is calculated by comparing the CPU execution time to the GPU execution time for each data size:

Number of Cities	Speedup (CPU/GPU)
5	0.142
6	0.206
7	0.206
8	0.209
9	0.402
10	2.613
11	23.980
12	58.596
13	>51.788

Theoretical Speedup

The theoretical speedup using GPU is derived from the potential of parallelizing factorial computations. Given that GPU can handle many threads concurrently, the speedup is ideally proportional to the number of threads and the efficiency of the implementation. For n cities, theoretical speedup could approach $O(n!/p)$, where p is the number of GPU cores available.

Discrepancy Between Theoretical and Actual Speedup

The actual speedup is below the theoretical maximum due to several factors:

1- Kernel Launch Overhead: The overhead associated with launching kernels can diminish speedup for smaller problem sizes.
2- Memory Transfer Latency: Transferring data between CPU and GPU memory can introduce delays.
3- Thread Divergence: Different threads may take different execution paths, leading to inefficiencies.
4- Synchronization Overhead: to calculate the global minimum

To achieve a better one :-
Optimize Memory Transfers: Minimize data transfers between CPU and GPU.
Increase Parallelism: Utilize more threads and blocks efficiently.
Reduce Synchronization: Use atomic operations or redesign algorithms to minimize locking.
Comparison to Open-Source Peers
Comparing the GPU implementation to open-source solutions such as those available in PyTorch or TensorFlow, which utilize optimized libraries for parallel computations:

PyTorch/TensorFlow: These frameworks offer highly optimized GPU operations for similar problems. They might achieve better performance due to advanced optimization techniques and extensive use of batched operations.
Custom CUDA Implementation: While our implementation shows significant speedup over CPU, open-source libraries might still outperform it due to better memory management and optimized parallel computation strategies.

In conclusion, while the GPU implementation demonstrates significant performance improvements over the CPU, especially for larger problem sizes, there is room for optimization. Leveraging more sophisticated parallel programming techniques and reducing overheads can bring the performance closer to theoretical expectations and open-source alternatives.