Leveraging the Power of CUDA: The Traveling Shopper Problem

Kimberly Chang kc875 165009475

Abstract

The Traveling Salesman Problem is a problem with well-known and well-defined methods to determine a solution. This paper introduces the Traveling Shopper Problem, which adds more constraints to the Traveling Salesman Problem. This paper implements a solution to the Traveling Shopper Problem, which is coded fully serially in C++ and fully parallel in CUDA.

Table of Contents

1. Introduction and Related Work

The Traveling Salesman Problem involves finding the shortest path to visit all nodes in a fully-connected, undirected, weighted graph. The Traveling Shopper Problem introduces a list of items that each node sells, a “home” or domicile node, and a shopping list. Rather than visiting each node like the Salesman, the goal of the Shopper is to obtain all items on their shopping list while limiting their travel (i.e., traveling the shortest path to do so). This can result in the Shopper not visiting every node.

The Traveling Shopper Problem is similar to the Traveling Purchaser Problem[1], but the Traveling Purchaser Problem has additional constraints. In the Traveling Purchaser Problem, the items also have prices at each store, and the Purchaser wants to minimize both travel cost (i.e. how long the path the Purchaser travels is) and purchase cost (i.e. the amount of money spent purchasing items).

The Traveling Shopper Problem is a median between the Traveling Salesman Problem and the Traveling Purchaser Problem. It is more constrained such that most, if not all, approaches to the Traveling Salesman Problem, will not work, but it is not so constrained that there could be no objective best path, like in the Traveling Purchaser Problem, where the solution is a balance of minimizing travel cost and purchase cost.

The Traveling Shopper Problem can be applied to real life situations. Like the Traveling Purchaser Problem, this problem can be used for real life shopping situations. The Traveling Shopper Problem can be considered as a subset of the Traveling Purchaser Problem, where either (1) all prices of an item are the same at every store where it is sold, or (2) the Shopper/Purchaser does not care about the price of items. The Traveling Shopper Problem can also be applied to some video games, specifically those where the player must obtain multiple items at different locations, especially if items can be obtained at multiple locations.

The goal of this work is to leverage the power of CUDA to parallelize a solution to the Traveling Shopper Problem to improve runtime.

2. Methods

2.1. Resources

For this project, the code was run on a Linux machine. The CPU for this machine is an AMD Ryzen 7 3800X 8-Core Processor, and the GPU for this machine is a NVIDIA GeForce RTX 2080 Ti.

2.2. Implementation

The implementation of the Traveling Shopper Problem runs in several parts. All parts are implemented both serially in C++ and parallel with CUDA (with a C++ base).

The first part of the implementation involves problem generation. First, an adjacency matrix is generated to represent the fully-connected, undirected, weighted graph. The distance between each node is set to be between 10 and 110 inclusive. Then, for each item on the shopping list, a list of shops that sell that item is randomly generated. Currently, each list will have the same number of shops.

The second part of the implementation involves solving the problem, which occurs in 3 steps. The first step is to generate all possible permutations of paths such that every item on the shopping list can be obtained by visiting nodes; this occurs by first generating unique combinations of nodes based on the item lists, and then generating permutations of each of the combinations. The second step is to determine the lengths of each of these paths. The last step is to determine the/a shortest path based on the length of each path.

2.2.1. Serial Implementation

2.2.1.1. Problem Generation

Generating the adjacency matrix takes $O(n^2)$ time, where n is the number of nodes in the graph. To create the adjacency matrix, a nested for loop is used. For each node, the distance between it and its neighbors must be generated. For a node and itself, this distance is 0, and for a node and another node, this distance can be anything within the range of 10 to 110 inclusive.

The first node determines the distance between itself and all nodes. The second node determines the distance between itself and all nodes except the first node. This result in $\frac{n*(n+1)}{2}$ generations of distances, which then simplifies to $O(n^2)$.

The adjacency matrix is stored as an `int *`, which is used to represent a 2D adjacency matrix of size n^2 (where n is the number of nodes in the graph) in row major order. Element `(i*n + j)` is used to represent the distance between node `i` and node `j`.

Generating the item lists takes $O(m * p)$ time, where m is the number of items in the shopping list, and p is the number of nodes per item. The item list is stored as a vector of integer vectors (`vector<vector<int>>`) for ease of implementation for the generation of possible paths.

2.2.1.2. Paths Generation

Initially, p^m paths, where p is the number of nodes per item, and m is the number of items, are generated based on the item lists. When generating a path, it must add a node from each item list to the path. After generating a path, it checks it against previously added paths to ensure that the paths are unique.

Since no work is performed in parallel, ensuring that there are less paths at this first section will likely result in less runtime because there will be less permutations to generate and check. The runtime is roughly $O((p^m * m)^2)$, where p is the number of nodes per item, and m is the number of items on the shopping list.

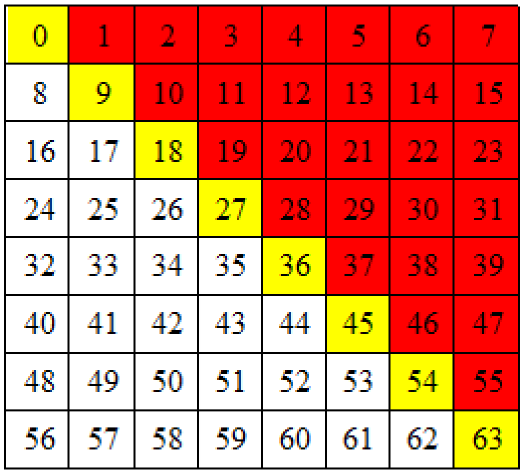
From the unique paths generated in the previous section, permutations of each path are generated to create every possible path such that all items on the shopping list can be obtained from the nodes on the path. Since for each path generated in the previous section, there are $m!$ permutations, there will be at most $p^m * m!$ paths after this section. Since this section generates all permutations for each path generated in the previous section, the runtime is roughly $O(p^m * m!)$, where p is the number of nodes per item, and m is the number of items on the shopping list.

2.2.1.3. Paths Lengths and Shortest Path

Calculating the length of each permutation takes at worst $O(p^m * m! * m)$ time, where p is the number of nodes per item, and m is the number of items on the shopping list. For each path, it must go through the whole path length, whose length is the number of items. The lengths of each path are stored in a 1D integer array of size $p^m * m!$.

Determining the shortest path takes at worst $O(p^m * m!)$ time, where p is the number of nodes per item, and m is the number of items on the shopping list. This is done by going through the array generated in the previous step, which is of size $p^m * m!$.

2.2.2. CUDA Implementation



🟨 0	🟥 1	🟥 2	🟥 3	🟥 4	🟥 5	🟥 6	🟥 7
8	🟨 9	🟥 10	🟥 11	🟥 12	🟥 13	🟥 14	🟥 15
16	17	🟨 18	🟥 19	🟥 20	🟥 21	🟥 22	🟥 23
24	25	26	🟨 27	🟥 28	🟥 29	🟥 30	🟥 31
32	33	34	35	🟨 36	🟥 37	🟥 38	🟥 39
40	41	42	43	44	🟨 45	🟥 46	🟥 47
48	49	50	51	52	53	🟨 54	🟥 55
56	57	58	59	60	61	62	🟨 63

Figure 1: A matrix representing the threads used for an example problem with 8 stores. Threads in red boxes do the most work: generating a number and applying it to 2 locations in the adjacency matrix. Threads in yellow boxes do some work: setting its equivalent location in the adjacency matrix to 0. Threads in colorless boxes are idle and do no work.

2.2.2.1. Problem Generation

For problem generation in CUDA, a thread is spawned for each element in the adjacency matrix and for each item in the shopping list.

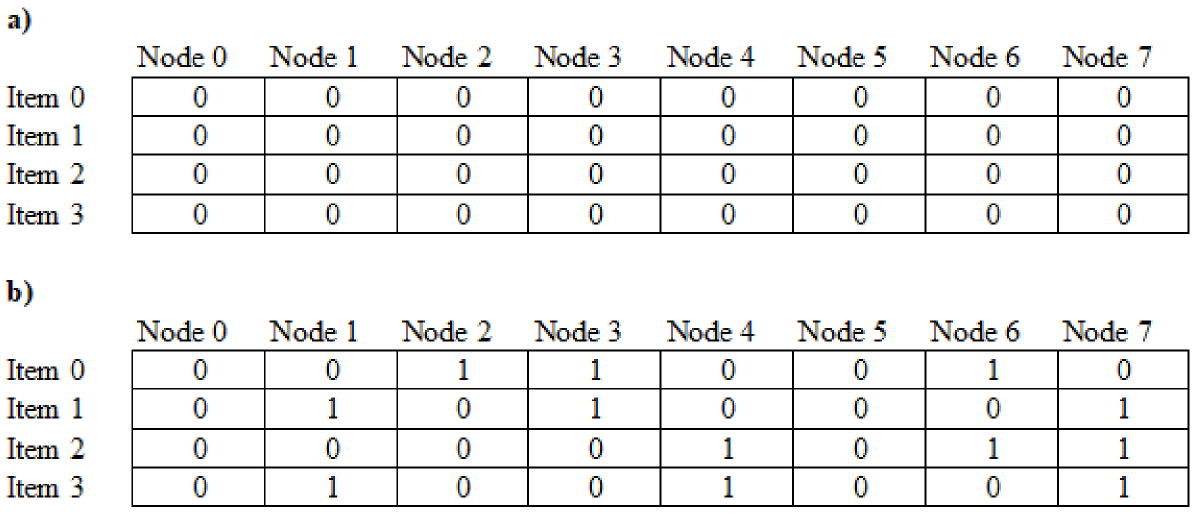
The threads for the adjacency matrix are generated in 1 block of $n \times n$ threads, where n is the number of nodes in the graph. Each thread is assigned to one element of the 2D adjacency matrix, which like the serial implementation, is stored as an `int *`.

Figure 1 shows an example where $n = 8$. In each box, a thread is associated with the corresponding element of the same number.

Since an adjacency matrix is symmetrical along its top-left to bottom-right diagonal, only the upper-right triangle of the matrix should actually do work. As a result, slightly less than half of the threads do no work; in Figure 1, these are the threads in the colorless boxes.

The threads handling elements on the diagonal do less work since they only fill their element with 0 because the diagonal of the matrix represents a node's distance to itself, which should be 0. In Figure 1, the threads handling the diagonal are the threads in the yellow boxes.

The threads in the upper-right triangle and not on the diagonal are the threads that do the bulk of the work; in Figure 1, these are the threads in the red boxes. Each of these threads randomly generates a number to represent the length between node `i` and node `j`, and then assigns it to both respective elements (i.e., `(i*n + j)` and `(j*n + i)`) in the adjacency matrix. One thread assigns the element to 2 locations in the matrix because having 2 threads work results in a communication overhead, which would likely cause the assignment to take longer than if just one thread performed the assignment to 2 locations.



a)

	Node 0	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7
Item 0	0	0	0	0	0	0	0	0
Item 1	0	0	0	0	0	0	0	0
Item 2	0	0	0	0	0	0	0	0
Item 3	0	0	0	0	0	0	0	0

b)

	Node 0	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7
Item 0	0	0	1	1	0	0	1	0
Item 1	0	1	0	1	0	0	0	1
Item 2	0	0	0	0	1	0	1	1
Item 3	0	1	0	0	1	0	0	1

Figure 2: Example boolean matrices for representing which item is sold at which store for a problem with 4 items and 8 stores. a) The matrix after being initialized to 0/false. b) The matrix after each thread picks 3 stores for its respective item.

The item lists are represented by a 2D boolean array that is stored as an `int *`, where element `(i*m + j)` is used to represent the `j`th generated node that sells item `i`. An example of a layout of the item lists is shown in Figure 2. Note that Node 0 in **b** is set to 0 for all items; this is because Node 0 cannot sell items since it is the domicile node. On the other hand, Node 5 just coincidentally ended up not selling any items on the shopping list.

For the item lists, a thread is spawned for each item. Each thread initializes its row/item list to 0 or false. Each thread then randomly picks p unique nodes, where p is the number of nodes per item. In Figure 2, $p = 3$. The nodes are guaranteed to be unique because while randomly picking nodes, the thread will repick if the thread has picked that node before.

2.2.2.2. Paths Generation

The path generation occurs in two sections. The first section generates p^m paths without ensuring they are unique. This results in more work overall, but allows the number of total permutations to be known earlier on. Moreover, if the amount of path permutations ends up being less than the total number of CUDA threads, the extra workload will likely not affect runtime. Paths are generated based on the base p equivalent. For example, thread `0` will generate a path using the first node in each item list.

The second section generates all permutations of paths generated in the previous section, resulting in $p^m * m!$ total paths. The item lists are represented by a 2D integer array that is stored as an `int *`, where element `(i*q + j)` is used to represent the `j`th node in path `i`, and where there are `q` paths.

2.2.2.3. Paths Lengths and Shortest Path

The lengths of the paths are calculated for all permutations. Repeat paths should not affect the determination of the shortest path because there will just be a repeat of the same value and path. These lengths are stored in a 1D integer array, where the index corresponds to the path index of the path in the 2D path array.

a)

Thread	Thread 0	Thread 1	Thread 2	Thread 3	Thread 0	Thread 1	Thread 2	Thread 3
Array	8	3	2	4	1	2	3	5
Array Index	0	1	2	3	4	5	6	7

b)

Thread	Thread 0	Thread 1	Thread 2	Thread 3
Minimum	1	2	2	4
Array Index	4	5	2	3

c)

Thread	Thread 0	Thread 1	Thread 0	Thread 1
Minimum	1	2	2	4
Array Index	4	5	2	3

d)

Thread	Thread 0	Thread 1
Minimum	1	2
Array Index	4	5

e)

Thread	Thread 0	Thread 0
Minimum	1	2
Array Index	4	5

f)

Thread	Thread 0
Minimum	1
Array Index	4

Figure 3: Demonstration of block reduction to find the minimum for an array with 8 elements using 4 threads.
a) The original array with each thread assigned to 2 elements in the array.
b) The minimum number and array index stored for each thread for round 1.
c) Block reduction/setup for next comparison (round 2).
d) The minimum number and array index stored for each thread for round 2.
e) Block reduction/setup for next comparison (round 3).
f) The minimum number and array index stored for each thread for round 3.

From the lengths, the shortest path is determined. Figure 3 shows the steps of the shortest path calculation for an array of length 8 using 4 threads. The shortest path calculation begins with spawning multiple threads (1,024 by default) and assigned a section of the path lengths array to each thread. A thread is not assigned a contiguous block of elements in the array; rather, it is assigned every n^{th} element, where n is the number of threads spawned. Figure 3a shows this type of assignment; for example, Thread 0 is assigned Element 0 and Element 4.

From its assigned section of the array, each thread determines the minimum of its section of the path lengths array and stores this value at the index corresponding to its thread index in a global array whose size is equivalent to the number of threads. The index of the minimum of each thread's section is stored in another global array similar the global array storing the minimum values. This process can be seen in Figure 3b, which shows each of the 4 threads having a minimum value and an array index stored.

After that, the first half of the threads compares its minimum with that of a thread's in the corresponding index in the other half of the global array and changes its value to the minimum. This process of halving the number of threads working continues until only the first thread (thread index 0) is comparing itself to the second thread (thread index 1) to determine the overall minimum. These steps can be seen in Figure 3c-f. Figure 3c and Figure 3e show the setup and comparison step. Figure 3d and Figure 3f show the result, albeit showing only the relevant section of the global array.

3. Results

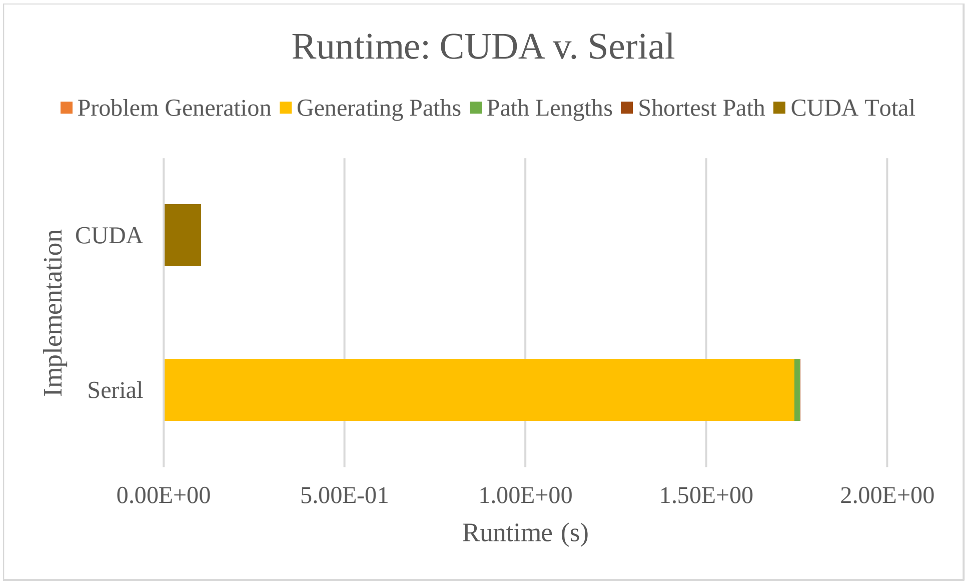


Figure 4: A graph comparing the average runtime of the sequential implementation and the parallel CUDA implementation.

	Runtime (s)	
	Serial	**CUDA**
Problem Generation	8.02E-06	1.03E-01
Path Generation	1.74E+00	
Paths Lengths	1.51E-02	
Shortest Path	1.10E-03	
Total Time	1.76E+00	1.03E-01

Table 1: A table showing the average runtime of each section of the sequential and parallel CUDA implementations.

Figure 4 compares the average runtime of the sequential implementation and the CUDA implementation, and Table 1 shows the exact times. The average runtime is calculated based on the average time taken for each section in a series of 100 runs, where the generated problem is not necessarily (and unlikely) the same. The generated problem always had 18 nodes in the graph, 6 items in the shopping list, and 3 nodes per item.

The part that actually ends up taking the longest time is the path generation since so many paths are generated. In a previous iteration of this project, the CUDA implementation only parallelized the second section of path generation, which is creating permutations, resulting in the runtime being cut in half. In this version of the project, the entire implementation was parallelized in CUDA, resulting in the runtime being cut by roughly 94% or a speedup of approximately 17.09.

4. Conclusion

This paper has introduced the Traveling Shopper Problem and presented a solution that is parallelizable using CUDA. It is possible that previous papers have worked on this problem and provided different solutions while calling it a different name since it is an obvious variant of the Traveling Salesman Problem, which has been widely discussed over the years.

Although some sections of the solution that are parallelized in CUDA result in a larger workload, the parallel nature of CUDA allows for this workload to be run simultaneous, so the larger workload does not significantly affect the runtime if at all. Using the Traveling Shopper Problem, this paper has shown how leveraging the power of CUDA can drastically improve the runtime of a program.

This project could be extended to involve real world data such that it could be used in a real life situation for an average consumer. The problem could be generated based on stores that a user frequents and a user's home address with lengths between nodes obtained from Google Maps data. Based on a shopping list, it could search online databases of the stores' websites to determine whether a store sells an item.

References

[1] Ramesh, T. (1981). "Traveling Purchaser Problem." *Opsearch,* 18, 78–91.