Accelerated Salesman

CS 432 : GPU Accelerated Computing
Habib University

Hamna Jamil, Wasiq Hussain

May 8, 2022

Contents

1 Abstract

GPUs are useful to compute CPU-intensive tasks which take up alot of RAM on traditional CPU. Using parallel computing, GPUs are able to give much faster performance as compared to CPUs. Our aim is to use capabilities of GPU to simulate Travelling Salesman Problem.

Keywords
travelling salesman problem, cuda api, 2-opt, GPU

2 Introduction

Travelling Salesman Problem refers to the problem of finding a shortest path in a complete graph such that each node is visited once and the path ends at the starting node. Consider the following example,[1]

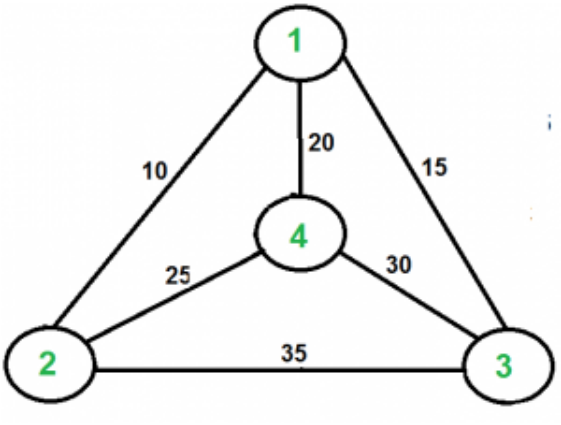

Figure 1: A TSP tour graph

With a little observation, we would know that a TSP tour in the graph is 1-2-4-3-1 which is the shortest path and covers all nodes. The cost of the tour is 10+25+30+15 which is 80.
There are a number of real-life scenarios where it is useful to find a TSP tour for example airline route planning where an aircraft takes off from a hub and has to come back to the starting point.
This problem belongs to NP-Hard class which means that there's no guarantee of finding optimal solution in given amount of time. Our aim is to reduce the runtime cost to solve this problem using a heuristic algorithm.

3 Literature Review

We will review the following available literature prior to our own experiments to better understand the problem.

3.1 Analysis of a High-Performance TSP Solver on the GPU

This paper published by Robsinson et.al in 2018. It uses cuda APIs along with 2-opt algorithm to improve naive brute-force approach. Pseudocode for underlying implementation is following:

```
1: BEST = 0 {each thread in a block performs the following}
2: while BEST ≠ -1 do
3:    BEST = -1;
4:    for i = 0 to cities − 1 up by num_threads do
5:       CITY = tour[i + threadID]
6:       NEXT = tour[i + threadID + 1]
7:       LAST = tour[cities]
8:       EDGE = (CITY,NEXT);
9:       for j = cities − 1 to i + 2 down by 1 do
10:         Load tour[j − num_threads...j] to SHARED
11:         for n = 0 to num_threads do
12:            OTHER = (SHARED[n],LAST)
13:            TMP = change(EDGE,OTHER)
14:            if TMP > BEST then
15:               BEST = TMP
16:            end if
17:            LAST = SHARED[n]
18:         end for
19:      end for
20:   end for
21:   BEST = max( BEST ) {max( ) gets the maximum decrease among all threads}
22:   perform_swap( BEST )
23: end while
```

Figure 2: Pseudocode for 2-Opt

3.2 A GPU IMPLEMENTATION OF LOCAL SEARCH OPERATORS FOR SYMMETRIC TRAVELLING SALESMAN PROBLEM

This paper by Fosin, Davidovic provides a deeper understanding of GPU implementation of TSP. We know that threads in a block share same memory so in order of reduce data retrieval cost, it is best to take full advantage of threads per block. Number of threads per block can vary according to different implementations and is subject to trial and error method.
2-opt is implemented in such a way that each thread evaluates exactly one pair. The variable i is the iterator through the current tour and j is the iterator of each customer's neighbours list. Each city has j+1 neighbours and for example each block contains j+3 threads, in block 0 threads 0,1,...j evaluate the new distances for the city pairs (0,0),(0,1),...(0,j) of the city c0 and the threads j+1, j+2 pairs ,and (1,1) of the city c1 as shown in Figure 3 [5].

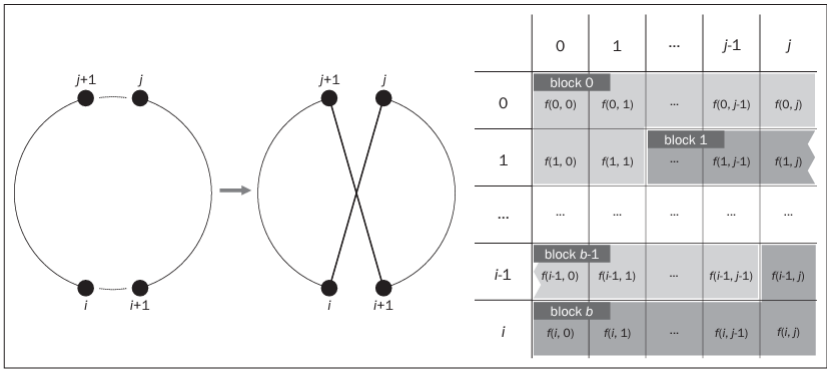

Figure 3: Left: 2-Opt; Right: Grid distribution

4 Methodology

After review of several research papers, we found that 2-opt is the most effective choice for different reasons. One is that it computes fast and deterministically improves. This means that we can increase the number of trials to get closer and closer to the optimal solution. We will discuss the details of this algorithm in the next part.
The 2-OPT technique was first proposed by Croes in 1958. 2-Opt works on the principle of nearest neighbor approach. This algorithm computes an initial solution and then iteratively improves it. The 2-opt algorithm works as follows: take 2 arcs from the route, reconnect these arcs with each other and calculate new travel distance. If this modification has led to a shorter total travel distance the current route is updated. The algorithm continues to build on the improved route and repeats the steps. This process is repeated until no more improve-

ments are found or until a pre-specified number of iterations is completed [2]. Our implementation is a heuristic algorithm because we want it to produce approximated solutions in a reasonable time. Direct approach would be to try all permutations which is a brute-force search and it will take polynomial factor of $O(n!)$ where n is number of nodes. The running time for this approach will be around the factorial of the total number of nodes, hence this approach to get exact result becomes impractical even for a small number of cities.

5 Implementation Details

All results were executed and tested on Tesla K80 running cuda version 11.2, using Google colab service provided by Google. More details can be found in the figure 4 below .
Our hypothesis is that, using existing implementation of 2-opt, we can substantially improve runtimes by parallel programming and proven modifications which are more efficient.

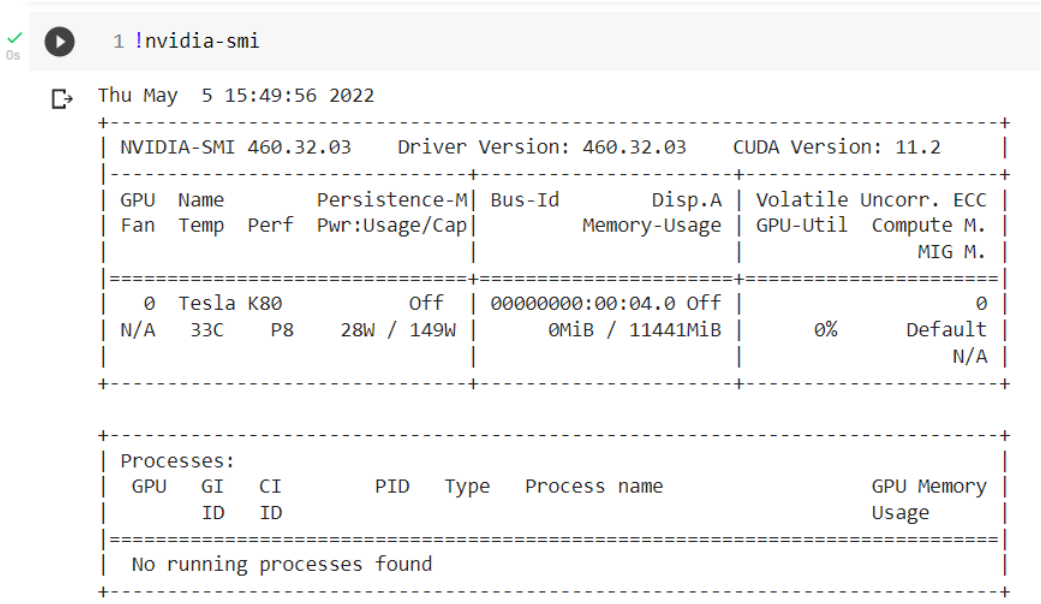

Figure 4: Nvidia System Management Interface

We use .tsp files from TSPLIB as data sets. The chosen datasets are wi29.tsp, qa194.tsp and pma343.tsp with 29, 194 and 343 nodes respectively. wi29.tsp and qa184.tsp have data from Western Sahara and Qatar respectively. They include each countries' cities along with the cost to travel between them. These datasets are sourced from National Imagery and Mapping agency. pma343.tsp is based

on VLSI data set provided by Andre Rohe of Forschungsinstitut für Diskrete Mathematik, Universität Bonn [3]. For our implementation, we use EUC-2D type as it allows graphs to be represented in euclidean x-y plane. Each data set is run 128 times to find the optimal solution.

We wrote CPU version first so that we can compare performance of the algorithm. Then we took the basic GPU code and made changes in it. We use several optimization techniques to improve the code. First one is use of unified memory. Everytime memory is accessed which is not resident on the device memory, it has to be migrated. This process is tedious and may cause delays. To avoid this we use prefetching techniques. For this purpose, cudaMallocManaged comes in handy because it automatically manages unified memory for you. Furthermore, from the literature review we found that selecting better choice for number of blocks and number of threads help in improving results. During the execution, blocks are referenced by (i,j) position in the Grid and then they are dynamically distributed to the streaming multiprocessors to be processed.

6 Resources

We have widely used Google Collaboratory for our code development. The language used in this project is Cuda C. The basic GPU code was taken from from an existing Github repository and changes were made afterwards. Links to source codes are given below;
Main Project File which includes CPU and Optimized GPU code:
TSP-CPU/GPU.ipynb
Reference Github Repository:
`https://github.com/tuomasr/cu-tsp`

7 Results

On testing different datasets against the CPU implementation and optimized GPU implementation, we found the following results:

Data set	nodes	CPU version(msec)	optimized GPU version(msec)	speedup
wi29.tsp	29	28.174	6.589056	4.275
qa194.tsp	194	12053.872	939.165955	12.834
pma343.tsp	343	68266.563	4575.22998	14.92

We can clearly see that using GPU has been a lot beneficial. The difference is small for small data sets but as we increase the size of our data set, the gap in performance increases a lot and the runtime on GPU is much lower than of CPU. The parallel computing for each iteration of finding tours, has greatly

affected the overall performance and running time of the same algorithm.

Data set	nodes	original GPU version(msec)	optimized GPU version(msec)	speedup
wi29.tsp	29	6.290368	6.589056	0.954
qa194.tsp	194	986.424377	939.165955	1.0503
pma343.tsp	343	4942.183105	4575.22998	1.0802

To highlight the difference made by improvements in code, we also performed tests on same datasets against original GPU code. Notice that there's slight improvement in the optimized version as we slowly increase the size of dataset. As expected, for a very small size, original code is in fact relatively faster but again as we increase the dataset size, we start to see an improvement in GPU times which will be even more evident in huge datasets. The use of Curand library to get random integers also contributed in producing a more optimal solution than normal C library rand () function in host code. At last, multiple optimization techniques like Unified Memory etc has given a good boost to the overall performance of the code. Even memory access time has also decreased due to less memory operations upon moving data from local memory to device memory.

To conclude, all three tests are plotted on the following graph,

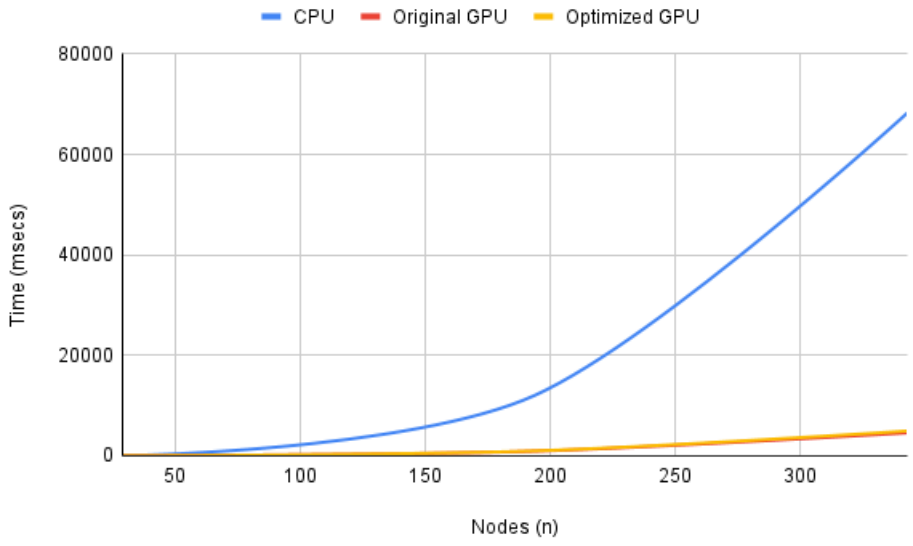

Figure 5: Comparison of Performance; CPU vs GPU vs Optimized GPU

8 Future

For future improvements, Cuda Streams can be used to run iterations for finding best tour and then grid blocks and threads can be used for permutation calculations and swapping of data. This way, the overall process can be optimized much better and achieve less runtimes.
Along with this, testing on a better system with a better local GPU can help in testing huge datasets in less time and room for improvements will also increase.

References

[1] Traveling salesman problem (TSP) implementation. GeeksforGeeks. (2020, November 4). Retrieved May 8, 2022, from https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/

[2] Venhuis, M. (2019, April 22). Around the world in 90.414 kilometers. Medium. Retrieved May 8, 2022, from https://towardsdatascience.com/around-the-world-in-90-414-kilometers-ce84c03b8552

[3] National traveling salesman problems. (n.d.). Retrieved May 8, 2022, from http://www.math.uwaterloo.ca/tsp

[4] Robinson, J. A., Vrbsky, S. V., Hong, X., amp; Eddy, B. P. (2018). Analysis of a high-performance TSP solver on the GPU. ACM Journal of Experimental Algorithmics, 23, 1–22. https://doi.org/10.1145/3154835

[5] Fosin, J., Davidović, D., and; Carić, T. (2013). A GPU implementation of local search operators for symmetric travelling salesman problem. PROMET - Traffic and ;Transportation, 25(3), 225–234. https://doi.org/10.7307/ptt.v25i3.300