

Travelling Salesmen Implementation in CUDA C

V.Shiva Swaroop
Computer Engineering
NITK

Email: shivaswaroopvardaneedi@gmail.com

M.Sri Charan Reddy
Computer Engineering
NITK

Email: sreecharanmallu04@gmail.com

I. PROBLEM STATEMENT

The main aim of the project is to study parallelisation characteristics of the Travelling Salesman Problem using NVIDIA CUDA. The project should present a good overview of approaches that were taken by other people and justify that the approach that was chosen is suitable for parallelisation. The secondary aim is to show how GPU can outperform CPU by demonstrating the speedup gained. It is also desirable to study scalability characteristics of the implementation as it performs on different CUDA-enabled devices

The travelling salesman problem is a nondeterministic polynomial hard problem. Now a days every optimisation problem is tested on Tsp first. Ant colony approach is used for solving Travelling salesmen problem.

Suppose there are N cities The Travelling Salesman Problem describes a salesman who must travel between N cities. The order does not matter but he should finish where he had begun. Considering each city is connected to other close by cities by a road(it is the link between nodes) and each roads have distances attached(weights), then the salesman would want to keep both the travel costs, as well as the distance he travels as low as possible. Since he does not want to spend much time on travelling, therefore we need to find the sequence of cities to minimize the traveled distance

II. PROJECT EXECUTION PLAN

A. Implementation Framework

CUDA using NVCC compiler

B. Inputs

map generator is used for generation of sample input. We will use Ruby language for writing map generator code

C. Expected Outputs

Given a set of points and distances Travelling Salesmen Problem should provide the most efficient path between the points.

Serial version of TSP will be implemented in C. We will implement the parallel TSP algorithm given in the paper in CUDA. We will compare the output deviance values from both the implementations for correctness of the answer. Also, we will compare the execution time to prove that the parallel implementation is faster.

D. metrics to measure

TSP is a graph theory problem, which searches for the shortest cycle through a graph, or the shortest path that visits each of n cities exactly. Applied to TSP, each ant constructs its own solution according to a combination of cost, randomness and memory. Afterwards, the pheromone matrix is updated by one or more ants placing pheromone on the edges of its tour according to solution quality. There exist variants of ACO. In the maxmin ant system (MMAS), only the ant with the best solution is allowed to deposit pheromone and the pheromone levels for each edge are limited to a given range. The two predominant, basic parallelization schemes are parallel ants, where one process/thread is allocated to each ant, and the multiple colonies approach. As the number of colonies increases, the number of tours also increases. This is implemented using both sequential and parallel codes. Both the sequential code and parallel codes are executed on the same input data. We compare Execution time and the resulting values.

E. ACO approach

Applied to TSP, each ant constructs its own solution according to a combination of cost, randomness and a global memory. Afterwards, the pheromone matrix is updated by one or more ants placing pheromone on the edges of its tour according to solution quality. There exist variants of ACO. In the maxmin ant system (MMAS), only the ant with the best solution is allowed to deposit pheromone and the pheromone levels for each edge are limited to a given range. The two predominant, basic parallelization schemes are parallel ants, where one process/thread is allocated to each ant, and the multiple colonies approach

III. PROJECT TIMELINE

Week 1	26 Oct
Sequential C code implementation of Travelling Salesmen Problem.	
Week 2	1 Nov
Parallel CUDA implementation of TSP.	
Week 3	8 Nov
Compare the results obtained from both the data sets and execution time comparisons for different data sizes and try to compare them in term of performance of both the implementations as shown in the research paper.	

And also, Compare the results for parallel implementation with different number of blocks and threads in each implementation and draw a conclusion from the results obtained.

IV. WORK DISTRIBUTION

Implementation of sequential C code for TSP and research paper reading.
-charan(16CO228)

Implementation of mapgenerator and research paper reading.
-Shiva(16CO252)

We both combinely help each other in CUDA implementaion of TSP.

V. WORK COMPLETED

The GPU implementation on CUDA was carefully prepared and presented. The CPU implementation on Java was also completed and thoroughly documented. Both implementations were driven through experiments in order to obtain trends that were analysed in the Evaluation

VI. MOTIVATION

Massively parallel computation has for a long time been considered expensive and was often not affordable to ordinary people interested in performing extensive dataset computation. In present days our society is faced with a fast-paced market of supercomputing, so the general trend in the industry shows that there is high performance where there is a larger market of consumers. When looking for a practical non-trivial problem that is very interesting to tackle on a GPU, the Travelling Salesman Problem(TSP) was learned from the Graph Theory course. The main practical applications of this problem are mainly in logistics, but it also appears in planning, manufacturing of microchips, DNA sequencing and many more. This problem has for a long time been one of the most challenging optimisation problems in the algorithmic community, and this is still the case. Furthermore TSP is known to be NPComplete, so solving it is generally very hard unless a polynomial solution is found at some point in future. If it was possible to solve this problem efficiently, then there would be many interesting applications, because this problem is connected to all other NP-Complete problems(CLIQUE, IND-SET, KNAPSACK, etc). Currently there are very good heuristicbased solvers that can solve instances of TSP to optimality or approximate answer to a reasonable amount. The central motivation for this project came from a desire to study parallelisation aspects of the Travelling Salesman Problem, which is something that could potentially be very useful for practical purposes. It was decided that NVIDIAs CUDA-enabled series of graphic cards constitute a suitable platform that is suitable for demonstrating those parallelisation aspects.

VII. THEORETICAL BACKGROUND

Firstly, define a Hamiltonian cycle:

Given a directed graph G, a Hamiltonian cycle is a path that visits every vertex exactly once and comes back to the starting vertex.

An informal definition of the Travelling Salesman Problem that is widely accepted

Given a weighed directed graph G, find a Hamiltonian cycle of minimum Total weight.

More formally, define a distance matrix D (where $d_{i,j} \in \mathbb{N}$) of weights of all edges in the graph $G=(V,E)$. The solver is seeking to find a matrix U, where $u_{i,j} \in \{0,1\}$ and $\sum_j u_{i,j} = 1$ and $\sum_i u_{i,j} = 1$ (this is just to ensure that there is only one incoming and one outgoing edge at each vertex), such that $\sum_{i,j} d_{i,j} u_{i,j}$ is minimum. Note that the sum would be infinity in the case where there is no Hamiltonian cycle in the graph G.

The CUDA environment is similar to theoretical Parallel Random Access Machine(PRAM) environment. One can reason about parallelisation perspectives of TSP by examining that the best complexity class for parallelisation is NC (Nicks Class) problems. PComplete problems are so far known to be hard to parallelise, however it is still not known whether $NC=P$, in which case there would be ways of parallelising P-Complete problems efficiently. Therefore given that it is unlikely that $P=NP$, NP-Complete problems could also be inherently not suitable for parallelisation.

VIII. APPROACHING TSP

There are many ways of tackling this problem. The most successful ones are very complex, such as Concorde TSP Solver. It is based on a branch-and-cut approach and can only

Given a weighed directed graph G, find a Hamiltonian cycle of minimum Total weight handle symmetric problem instances. However, one can easily turn any asymmetric TSP instance into a symmetric one, just by doubling the number of vertices and doing some distance matrix manipulations.

It is desirable that any solution to TSP can handle any graph topology equally. This is driven by the fact that it is easy to come up with heuristics that perform extremely well on certain types of graphs, while fail on the others.

Most solutions fall into two categories: exact and approximation solutions

IX. EXACT SOLUTION

One can solve some instances of TSP by simply brute-forcing all possible paths in time $O(N!)$. This approach is parallelisable by dividing the search space into subspaces for each parallel processor. Such subdivision is subject to scalability and also ensuring that subspaces do not overlap. Another approach is to use dynamic programming. Held and Karp proposed a simple solution that is $O(N^2N)$.

The most effective approach is branch-and-bound which is a tree search technique that uses a bound heuristic which narrows down the search tree. Such exact solution method can also be coupled with Lin-Kernighan heuristic [11] to obtain an efficient exact TSP solver and also a good approximating solver that finds local minima.

An important thing to note is that any kind of branch-and-bound technique would involve a tree search, which is tricky in a parallel environment because of inherent control

dependencies. Pekny and Miller developed a parallel branch-and-bound algorithm for TSP on a data-flow computer which is not exactly similar to the target architecture for the work. The exact solution approach is therefore not examined in this study. Another reason is that CUDA is not capable of creating threads dynamically, in contrast to Javalike multi-threading environment. Harish and Narayanan studied various tree searching techniques with CUDA and their report shows unimpressive results.

X. APPROXIMATION APPROACH

The main algorithm used in this study is called Ant-Colony Optimisation(ACO) which was originally developed by Marco Dorigo . The idea is inspired by how real ants find food in nature. Every ant lays some amount of pheromone everywhere it passes. Ants move by sensing pheromone and cognitively preferring to head towards places where there is more pheromone. As a result, there is an efficient network of trails that ants use to signal where the food is.

In ACO for TSP, virtual ants move from vertex to vertex until they complete a Hamilton cycle. Their total distance covered is then calculated and pheromone values are updated on each edge of the covered path accordingly. Virtual ants use the following probabilistic function to randomly choose where to go next. This is called the state-transition rule:

$$p_{i,j} = \frac{(\tau_{i,j}^\alpha)(\delta_{i,j}^\beta)}{\sum (\tau_{i,j}^\alpha)(\delta_{i,j}^\beta)}$$

where

1. i,j is the amount of pheromone on edge (i,j) .
2. i,j is the desirability of edge (i,j) ($= 1/d_{i,j}$ to make shorter edges more desirable).
3. α and β are parameters to control the influence of i,j and i,j respectively. These were set to 1 in this study because of problems related to losing floating point number precision. However, it would also be interesting to study how these parameters affect the operation of the algorithm in a separate study.

Pheromone update is normally done using the following formula (applied by each ant as it visits edges of the graph). This is also called the local update rule

$$i,j = (1 - \rho)i,j + 0$$

where

1. ρ is the local rate of pheromone evaporation. Set to be 0.5 in this work, but it would be interesting to have a separate study on this one as well
2. 0 is an approximation pheromone characteristic. Controls how much pheromone is laid.

XI. GPU IMPLEMENTATION

The GPU implementation contains a few files. Functionalities in files are divided as follows:

map-generator.rb : A ruby program used to generate map with desired cities (distance between one with all other are generated along with hormone).

ants.c : serial code for travelling salesman problem in c language.

Makefile : To make the files build comfortably without all commands. just give a make.

Parallel-ants.cu : parallel code implementation in CUDA.

Map100.txt : generated map from the map-generator.rb

XII. HOST CODE

The host code is mainly located in the parallel-ants.cu file. This setup routine ensures that a CUDA-enabled device is present in the system and will alert if no such device exists. Another part of the initialisation routine is the host copying data to the GPU. This is done by calling special functions like cudaMalloc and cudaMemcpy for allocating memory on the device and then copying data of appropriate size from the host to the device. The data that are allocated and copied include: distances array(delta), pheromones array(tau), random numbers array(R), problem size integer(N), result integer(best).

At the point when all the data has finished copying, the device is ready to run the core part of the program. the host side should simply launch the ACO routine (or kernel, in CUDA terms) on GPU for K number of times. Here, K can be any value that satisfies the needs of the study as it is only significant for cases when the algorithm is failing to find the right answer, so increasing K will give some extra time. The value for K that was used for this study is 2500. Calling the ACO kernel (which is located in parallel-ants kernel.cu and defined as external function at the host side) simulate

```
simulateants <<< BLOCKS,THREADS >>>
(antsd, stated, distancesd, hormoned, THREADS);
```

XIII. ANT ACTIVITY DIAGRAM

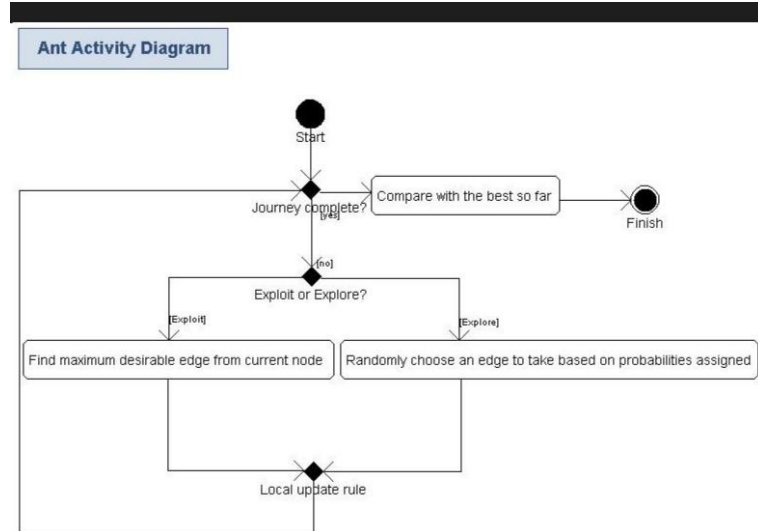


Figure : Activity Diagram of an ants actions inside of the simulate kernel.

After choosing the next edge to take, its weight is added to aggregator variable sum. Also local update rule (Formula 2) has to be applied to the corresponding edge. The procedure gets repeated again until $n-1$ edges are collected and all

vertices are visited. The final step to make would be to pick the edge connecting the starting vertex and the final vertex in order to form a cycle. The best so far (d best, stored in device memory for all threads to share) is then compared to the aggregate sum of the whole trip and updated if needed. One minor problem is caused by the fact that CUDA does not allow dynamic memory allocation inside of a kernel. This forces one to create static-size arrays inside of a program. The visited array which stores the current path must be of size n. Because n is a parameter, it is only possible to create a static-size visited array. It was decided that a size large enough would be chosen to accommodate for all problem instances that would be met in the course of this study.

XIV. RUNNING TIME

The idea of this experiment is to observe how the program performs with different problem instances

cities	CPU	GPU
25	52.05 sec	0.5 sec
50	63.90 sec	5.46 sec
100	57.86 sec	6.03 sec

XV. CONCLUSION

Parallelisation is actually an improvement. The book also claimed that the optimal number of ants to use should be either 64 or equal to the number of vertices in the graph. It was shown quantitatively that increasing the number of ants improves convergence, while too many ants cause diminishing returns as an underlying reason.

The project aims were met, however there is also one potential aim that is desirable, but deliberately was not initially included because it could have been too hard to achieve. That aim is the actual real-world applicability of the project. In reality, it might be useful to use a program such as developed for this study in businesses of large logistics companies or any other companies that depend on optimisation problems heavily. A real-world problem may consist of thousands of nodes with all sorts of externalities that the program developed here would not be able to solve. It is known that there are other efficient implementations that perform much better than the approach developed in this study. Of course, those implementations are likely to use very complicated heuristics, where this project only uses a few. Another reason could be that the algorithm chosen (ACO) is suitable for TSP, but must be implemented differently. It is the case though that ACO would perform reasonably well on constantly changing graphs. A more disappointing issue is that running times of ACO are non-deterministic and rely heavily on the quality of random numbers produced. On the other hand, TSP is an NP-Complete problem, so it would be ambitious to expect too much from this approach and getting to a 20

XVI. REFERENCES

[1] M. Dorigo and T. Stutzle. A short convergence proof for a class of ant colony optimization algorithms. IEEE

TRANSACTIONS ON EVOLUTIONARY COMPUTATION, VOL. 6, NO. 4, 2002.

[2] M. Dorigo and T. Stutzle. Ant Colony Optimization. MIT Press, 2004.

[3] M. Hahsler and K. Hornik. Tsp infrastructure for the traveling salesperson problem. Journal of Statistical Software, 2007.

[4] P. Harish and Narayanan P.J. Accelerating large graph algorithms on the gpu using cuda. 2007.

[5] M. Held and R.M. Karp. A dynamic programming approach to sequencing problems. Journal of SIAM, 1962.

[6] S. Lin and B.W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. Operations Research, 1973.

[7] J.F. Pekny and D.L. Miller. A parallel branch and bound algorithm for solving large asymmetric traveling salesman problems. Proceedings of the 1990 ACM annual conference on Cooperation, 1990.