Shiva swaroop. V – 16CO252
Sree Charan. M – 16CO228
National Institute of Technology, Karnataka

Computer Archietecture Project

Parallelising the Travelling Salesman Problem on NVIDIA R CUDA architecture

November 15 , 2018

Original Aims of the Project

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.

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.

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 NP-Complete , 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 heuristic-based 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 NVIDIA's CUDA-enabled series of graphic cards constitute a suitable platform that is suitable for demonstrating those parallelisation aspects.

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.

follows:

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

One can clearly see that, in total, there are $O(N!)$ paths in every problem instance (simply taking permutations of all vertexes). This means that the lower bound for TSP is around $O(NlogN)$. However, no approaches have been found so far that can solve any classical TSP problem in polynomial time. The problem is clearly in class NP (non-deterministic polynomial), and moreover it can be shown that this problem is also NP-hard [3, 6]. Hence this problem lies in the NP-Complete set of problems and can be trivially reduced to the SAT (boolean formula satisfactiability) problem which is known to be ultimately NP-hard according to the Cook's theorem.

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 (Nick's Class) problems [10]. 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.

Approaching TSP

There are many ways of tackling this problem. The most successful ones are very complex, such as *Concorde TSP Solver* [2]. It is based on a branch-and-cut approach and can only

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.

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 [9] proposed a simple solution that is $O(N^2 2^N)$.

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 [12] 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 multithreading environment. Harish and Narayanan [8] studied various tree searching techniques with CUDA and their report shows unimpressive results.

Approximation approach

The main algorithm used in this study is called *Ant-Colony Optimisation*(ACO) which was originally developed by Marco Dorigo [5]. 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*:

$$p_{i,j} = \frac{(\tau_{i,j}^{\alpha})(\delta_{i,j}^{\beta})}{\sum(\tau_{i,j}^{\alpha})(\delta_{i,j}^{\beta})} \quad (1)$$

where

- $\tau_{i,j}$ is the amount of pheromone on edge (i,j).
- $\delta_{i,j}$ is the desirability of edge (i,j) (= $1/d_{i,j}$ to make shorter edges more desirable).
- α and β are parameters to control the influence of $\tau_{i,j}$ and $\delta_{i,j}$ 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*:

$$\tau_{i,j} = (1 - \rho)\tau_{i,j} + \tau_0 \quad (2)$$

where

- 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.
- τ_0 is an approximation pheromone characteristic. Controls how much pheromone is laid.

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

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. As shown on Figure 3.1, 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_ants <<< BLOCKS, THREADS >>> (ants_d, state_d, distances_d, hormone_d,
THREADS);
```

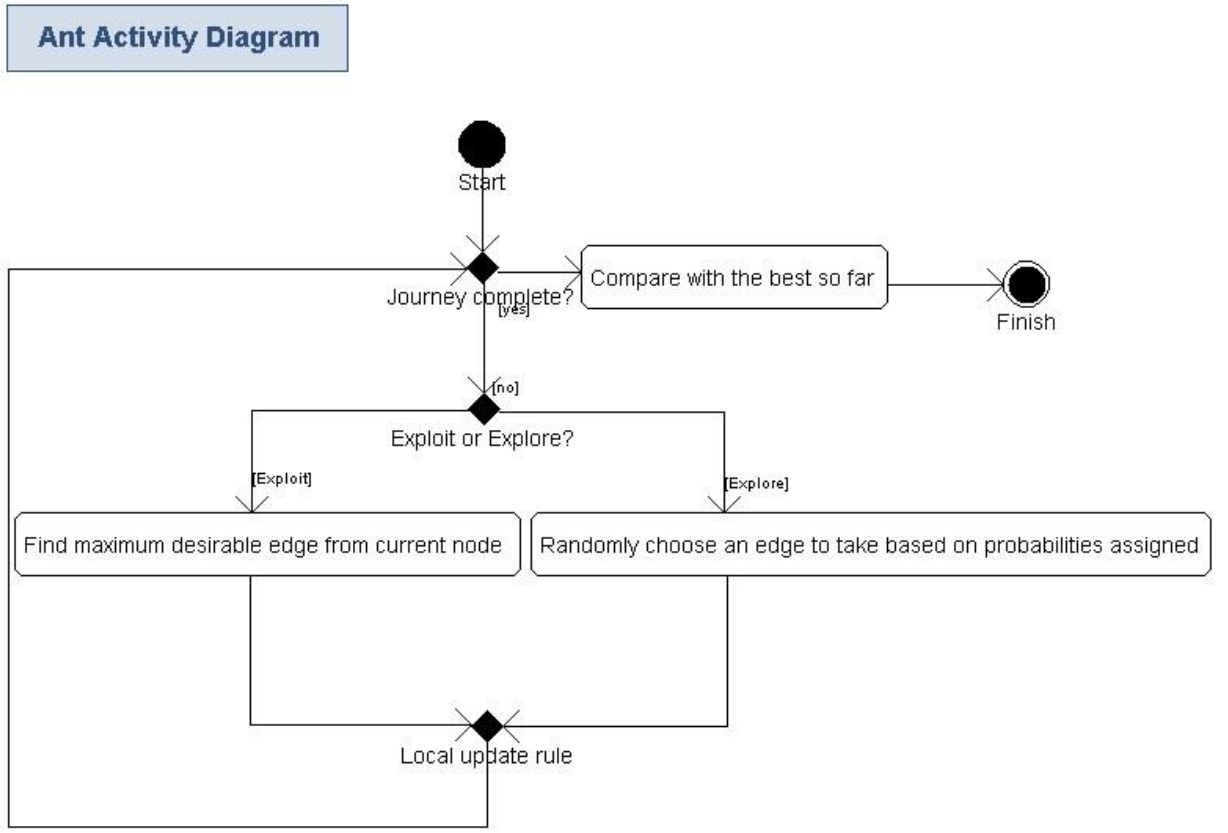

Figure : Activity Diagram of an ant's actions inside of the simulate kernel.

- using Formula 2.1 iteratively sum up probabilities for each valid edge until the sum is larger than the random number taken
- the final edge is the next transition to take in accordance with probabilistic choice

After choosing the next edge to take, its weight is added to aggregator variable sum. Also local update rule (Formula 2.2) 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.

Running time

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

No. of CITIES	**Execution time(CPU)**	**Execution time(GPU)**
25	52.05 sec	0.5 sec
50	63.9 sec	5.46 sec
100	57.86 sec	6.03 sec

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% accuracy is still very hard to guarantee on every single problem instance.

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.