Parallelizing The Traveling Salesman Problem (TSP) on Multi-GPU system

Vikas Patidar
New York University
vp1274@nyu.edu

Praveen Prakash Oak
New York University
praveenoak@nyu.edu

Abstract

In this paper, we describe a highly optimized approach to parallelize the 2-opt approximation algorithm for finding the solution to Travelling Salesman Problem(TSP). Travelling Salesman problem is a widely popular problem in computer science with applications in transport, genetics, combinatorics, mechanics and supply-chain domains. The proposed implementation[1] is highly efficient and parallel for executing it on GPU. We show that for medium size(500-1000) problems the Multi-GPU based solution provides speedup of around 60 over CPU based solution.

1 INTRODUCTION

Travelling Salesman Problem(TSP) is one of the most explored optimized problem in combinatorics. Let G be an undirected graph with V as vertex set, E as edge set and W as the edge-weight set. Hamiltonian cycle of G is a cycle in G which starts at some source vertex s and traverses all other vertices exactly once before returning back to the source vertex. TSP is a special case of finding the minimum cost hamiltonian cycle. In TSP each vertex is a city and edge-weight w of edge(u,v) is the cost of travelling from city u to city v. The goal is to find the minimum cost hamiltonian cycle on this graph.

As finding a solution to either of Hamiltonian Cycle or TSP is NP-Hard, finding exact solutions of medium to large TSP problems take years. This is why various heuristics based approximation algorithms are developed to find solutions as close as possible to the optimal. In this work, we make use of 2-opt algorithm in the approximate solution.

With their highly parallel structure, graphics processor units (GPUs) are specifically designed to perform data parallel computation. Because of the architectural differences between the central processing units (CPUs) and GPUs, the throughput-oriented and data parallel applications with intense arithmetic operations, are very well suited to GPUs compared to the CPUs. GPUs have the capability to accelerate algorithms that require high computational power. Thus, can significantly reduce the execution time of such computations by performing these intense calculations in parallel manner.

While approximate solutions reduce the execution time of otherise uncomputable NP-Hard problems like TSP problem, accuracy and quality of these solution is also an equally important factor. Local search algorithms like 2-opt are computationally infeasible when they are implemented on CPU because it evaluates all edge exchanges on the tour to find the exchange that reduces the tour cost by maximum amount.Performing this computation on CPU is only possible for small size problems. GPUs which have data parallel structure can perform these intense computations in parallel. On the other hand, the design of the parallel implementation plays a crucial role in achieving effective utilization of the GPU resources, thus optimizing the system performance.

2 Literature Survey

(Melab et al., 2009) used a hybrid approach to parallelize TSP problem. They used GPU for each iterative computations, where for each 2-opt exchange, they find solutions to TSP problems with single kernel. All other operations were executed on CPU.

In 2-opt algorithm there are mainly four steps: selecting a pair of cities to exchange in the current cycle, evaluating the candidate cycle, selecting the smallest cost cycle across all pair of exchanges and updating the solution. (Ermiş and Çatay, 2017) created the neighborhood on the CPU and trans-

[1] https://github.com/praveen-oak/parallel-tsp

ferred it to GPU every time. A lot of information was copied from CPU to GPU in this approach. (Rocki and Suda, 2012) addressed this drawback. They utilized an explicit formula to explore the neighborhood by assigning one or several moves to a thread. Evaluation stage was done on GPU side.(O'Neil and Burtscher, 2015) presented random restart hill climbing with 2-opt local search for TSP in parallel. (Dorigo and Gambardella, 1997) use Max-Min Ant System(MMAS) algorithm (Stützle and Hoos, 2000) for solving TSP problem.

Using advanced CUDA techniques, we explore various possible approaches including shared memory, constant memory, streams, atomic operations. By taking various trade-offs into account we build a highly parallel and optimized solution. Along with this we implemented dynamic and static GPU allocation according to compute availability and problem size.

3 Proposed Idea

The methodology was to first implement the sequential version of the 2-opt approximation algorithm. After which, we looked at the iterative and independent portions of the solution, i.e. the portions which are parallelizable. The 2-opt algorithm works by starting with an arbitrary Hamiltonian cycle with a fixed start and end node. At each iteration, the algorithm swaps each pair of nodes in the cycle to find the swap that best improves(decreases) the cost of the cycle. After running through all possible swaps(total of n(n-1)/2), the best swap is made and the next iteration starts. This iterative process continues until none of the swaps improves the cost any further, at which point the algorithm stops. The above process is repeated with each node and the start and end of the cycle. In this process, there are

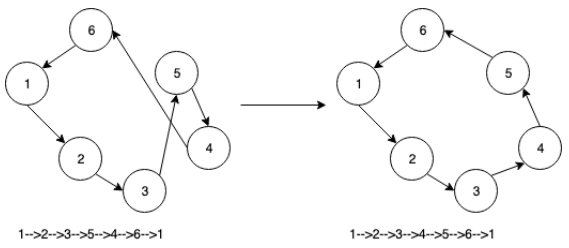

Figure 1: A 2-opt move on the TSP cycle

two operations which are independent, and hence parallelizable. Firstly, for a given cycle, the improvement by swapping can be calculated independently. However, the next iteration of finding the next best swap cannot begin before the cycle is updated after finding the best possible swap in the current cycle. The amount of possible parallelization is limited by this. Secondly, the process of finding the best permutation for each node and the start and end of the cycle can be parallelized.

Using the above two observations, we developed a parallel algorithm, with one GPU kernel which parallelly calculates and reports the best possible swap. Inside each GPU, shared memory is used to find the best improvement swap for each block. The best value in each block is stored in global memory, from which the best overall swap can be calculated. This kernel is called repeatedly from the CPU until the kernel reports no further improvement. And further, the best permutation for each node as the start node is calculated in parallel by pthreads. The decision of using pthreads here was to allow for another higher level of parallelization with the ability to scale smoothly for multi GPU systems.

In the process of developing the solution, we attempted multiple different variations of the final solution, based on the profiling results we observed. One alternative technique was to use constant memory to store coordinates of nodes and calculate the distances on the fly in the GPU(storing distances in constant memory is not possible due to the O(n2) space requirement.) However, as the profiling and occupancy results showed that the algorithm is compute bound and hence any improvement in memory latency or bandwidth does not improve the speed and in fact decreases it because of the additional repeated computation involved. A second alternative approach was to reduce memory transfers from CPU to GPU by updating the current cycle in the GPU using a second kernel. Again, no improvements were observed as the algorithm is compute bound.

4 Experimental Setup

All experiments for this effort were carried out on the High Performance Cluster at NYU CIMS. The details of the CPU and the GPU(s) used are tabulated in Table I and II.

Multi GPU experiments involved GPUs of the same kind as listed above. All the experiments were run 5 times each and median values were reported.

Property	Value
Device	CPU
Model	Intel Xeon E5-2650v2@2.6GHz
Max Frequency	3.4GHz
CPU(s)	16
Thread/cores	1
L1/L2/L3 cache	32/256/20480K
Compiler	GCC

Table 1: CPU Configurations

Property	Value
Device	GPU
Model	GeForce GTX Titan Z
Compute Capability	3.5
Max thread per block	1024
Max threads per SM	2048
Constant Memory	65536 bytes
L2/Shared Memory	1572864/49152 bytes

Table 2: GPU Configurations

5 Benchmarks

We used data from two popular TSP data libraries found online. The first was TSPLIB and the second was TSP Data Library from the University of Waterloo. We used the time.h C library to calculate runtimes and nvprof and nvidia-smi tools to profile the experiments.

6 Results & Discussions

6.1 Speedup

We ran the sequential version(written in C and compiled on GCC) with a parallel version(written in CUDA and compiled on NVCC), on the machines and benchmarks described above. In general, we observed a speedup of 50-60x on the multi GPU system(4 GPUs) versus CPU. The speedup increased progressively from almost nothing to a maximum 61x on observed on the gr666 dataset, as expected. Further, the sequential version failed to complete on the larger mu1979 dataset, and the process was killed by the operating system after running for 1283 minutes(21.38 hrs). Also, there was no difference in the result reported by the CPU and GPU versions(100% accuracy with 100% precision). The results are depicted in Fig 2 & 3.

These charts represent the gist of the results, detailed data gathered is available in tabulated form in the supplementary material[2].

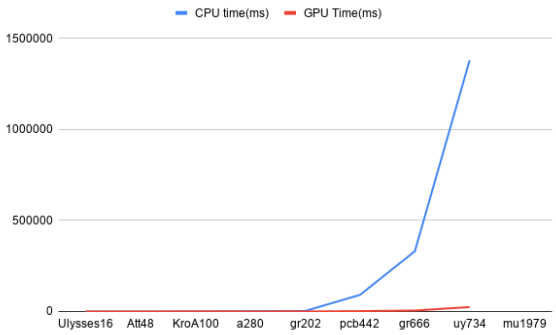

Figure 2: Execution Time vs Problem Size

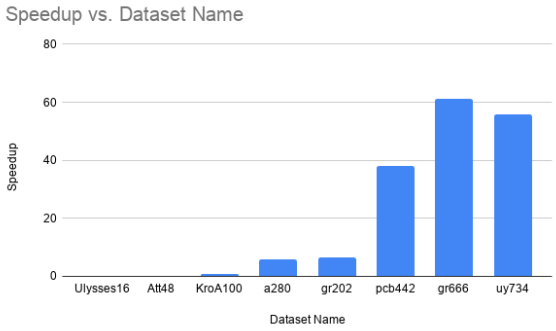

Figure 3: Speedup vs Problem Size

Devices	Time(ms)	CPU Time(ms)	Speedup
1	93201.24	1381234	14.82
2	48105.81	1381234	28.71
3	33096.63	1381234	41.73
4	24482.07	1381234	56.42

Table 3: Speedup vs Number of Devices

6.2 Scalability

To test the scalability of the algorithm, we tested the parallel version of the algorithm on the uy734 dataset. We observed an almost perfect scaling efficiency of 100% when the number of devices were varied from 1 to 4. Further the occupancy on each device was identical, which showed that the algorithm did a good job in distributing the load equally. The results of scalability experiments are shown in Fig. 3 and Table III.

6.3 Occupancy and Overhead

We used nvprof to time profile the algorithm and investigate where the algorithm spent most of its

[2]https://github.com/praveen-oak/parallel-tsp/blob/master/report/GPU%20Supplement%20Info.pdf

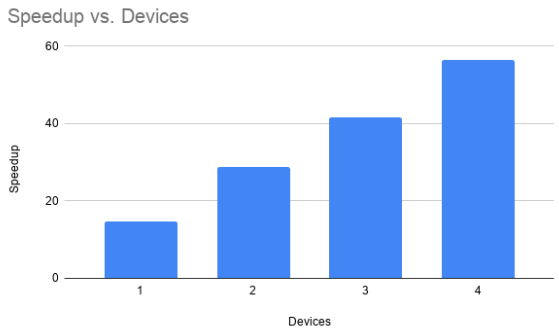

Figure 4: Speedup vs Time(milliseconds)

Dataset Name	Dataset Size	Kernel Time(%)	Comm Overhead Percent(%)	Kernel Time-Overhead Ratio	Avg Occupancy
Ulysses16	16	82.42	17.58	4.69	0.75
Att48	48	83.75	16.25	5.15	4
KroA100	100	85.29	14.71	5.8	4
a280	280	90.95	9.05	10.05	17.5
gr202	202	88.28	11.72	7.53	35.5
pcb442	442	94.25	5.75	16.39	70
gr666	666	96.45	3.55	27.17	79
uy734	734	97.19	2.81	34.59	84
mu1979	1979	98.9	1.1	89.91	96

Table 4: Occupancy and Communication Overhead

runtime and look for potential bottlenecks. The ratio of kernel time(useful work) to communication overhead(memcpy DtoH and HtoD) time was calculated for each dataset. As the problem size got bigger, the ratio improved dramatically, which shows that the algorithm parallelized well. Further, the occupancy information shows that all 4 GPUs are almost fully occupied when the problem size gets large enough. The occupancy is almost even throughout the life of the program which shows that GPU resources are being fully and evenly used. Table IV, Fig 5 and Fig 6 illustrate the results of this experiment.

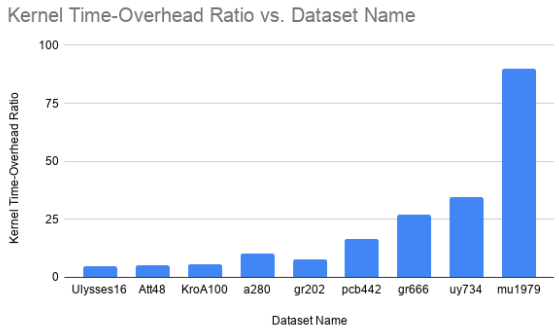

Figure 5: Kernel Time to Communication Overhead Ratio vs Problem Size

7 Conclusions

- The 2-opt algorithm is highly parallelizable and the algorithm we developed showed promising results for large problem sizes.

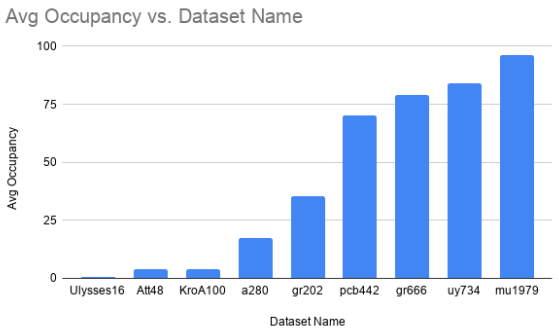

Figure 6: Avg Occupancy vs Problem Size

The algorithm was able to achieve 50-60x speedup over CPUs and performed well under extremely large problem sizes where the CPU failed to run in a reasonable amount of time.

- Further, the algorithm proved to be highly scalable as well, with the scaling efficiency being close to 100% as the number of GPUs moved from 1 to 4, provided that the problem size was large enough.

- The algorithm made good and judicious use of compute resources. It was not bounded due to communication overhead or setup time, but rather spent most of the run time in the GPU with close to 100% occupancy. Further, the load on the GPU remained close to constant over the entire runtime. This shows that the algorithm is highly optimized and tuned to run on parallel programming paradigms.

References

Marco Dorigo and Luca Maria Gambardella. 1997. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation*, 1(1):53–66.

Gizem Ermiş and Bülent Çatay. 2017. Accelerating local search algorithms for the travelling salesman problem through the effective use of gpu. *Transportation research procedia*, 22:409–418.

Nouredine Melab, El-Ghazali Talbi, et al. 2009. Parallel local search on gpu.

Molly A O'Neil and Martin Burtscher. 2015. Rethinking the parallelization of random-restart hill climbing: a case study in optimizing a 2-opt tsp solver for gpu execution. In *Proceedings of the 8th workshop*

on general purpose processing using GPUs, pages 99–108. ACM.

Kamil Rocki and Reiji Suda. 2012. Accelerating 2-opt and 3-opt local search using gpu in the travelling salesman problem. In *2012 International Conference on High Performance Computing & Simulation (HPCS)*, pages 489–495. IEEE.

Thomas Stützle and Holger H Hoos. 2000. Max–min ant system. *Future generation computer systems*, 16(8):889–914.