Implementing the Travel Salesman Problem using CUDA in GPU

Rachit Mehrotra
Dept. of Computer Science
New York University
rm4149@nyu.edu

Abstract—

The Travelling Salesman Problem (TSP) is a complex problem in combinatorial optimization that cannot be solved conventionally particularly when number of cities increase. The aim of this study is compare the effect of using two distributed algorithm (popular meta-heuristics techniques used for optimization tasks) which are Ant Colony Optimization (ACO) as a Swarm intelligence algorithm and Genetic Algorithm (GA). To solve the Travelling salesman problem in GPU, I used these algorithms after doing a survey of algorithms that solve TSP. These solutions were picked out of the possible GPU-friendly solutions presented, based on a list of criteria's on why they are better. Once they were implemented, they were compared to each other on the same criteria's. For Ant Colony Optimization, I studied the effect of some parameters on the produced results, these parameters as: number of used ants, maximum tours, and number of cities. On the other hand, I studied the chromosome population, crossover probability, and mutation probability parameters as well as the affect of number of cities, generations and population affect on the Genetic Algorithm results. The comparison between Genetic Algorithm and Ant Colony Optimization is accomplished to state the better one for travelling salesman problem. The results of comparison show that ant colony is better than genetic algorithm and it requires generally only a few lines of code.

Keywords—Travelling Salesman Problem, genetic, ant colony, optimization, parallelism, CUDA, GPU-friendly

I. Introduction

The travelling salesman problem (TSP) is a nondeterministic polynomial hard problem in combinatorial optimization and is the most studied optimization problem favorable among researchers. It is so important that almost every new approach for solving Optimization problems is first tested on TSP. Though it looks simple, it is one of the classical optimization problems, which cannot be solved by conventional mathematical approach. In this project, I have used two metaheuristic approaches- ACO and GA based on certain criteria's that make them better as described in later sections, for solving TSP. 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 link (e.g.: road) which has one or more weights (or the cost) attached, 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. In short, TSP is a graph theory problem, which searches for the shortest Hamiltonian cycle through a graph, or the shortest path (optimal route) that visits each of *n* cities exactly once. As the number of nodes increases, the number of tours also increases. With N cities, there are N! possible paths. The obvious "brute force" approach is to compare all N! paths to find the shortest. This is not feasible for large values of N. Rather than find an exact solution to TSP, what most do is find an approximate solution. An approximation algorithm is one, which finds a "good enough" solution to a problem, which may or may not be the best one possible.

As TSP has a search space, which grows at least exponentially to the number of nodes/cities, using GPU as general purpose computing devices, exhibits a lot of parallelization, which is a good usage for this problem to speed up optimization, which is why I used the GPU-friendly algorithms ACO and GA to solve TSP.

Idea of massive data parallelization to speed up algorithms to alleviate computational hardness date back to early 1990's. Some methods like swarm intelligence- ant colony optimization or population-based metaheuristics involve different types of parallelism at different levels of granularity. Others like neighborhood exploration in local search are inherently parallel. Speed comparison has been used as the most common way of justifying GPU implementation over CPU implementation. But there are more important aspects like utilization of GPU hardware and different algorithmic approaches on GPU because an algorithm organized in different way has variety of GPU implementations each using different GPU specifics such as shared memory. Each solution I used, presents its quality i.e how close to optimal it is and its speedup. Also, the consideration of how changing problem size affects the complexity overall.

In the next sections I discuss the background information in section 2, literature study in section 3, proposed solution in section 4, experimental setup in section 5, results in section 6 and conclusion in section 7.

II. Background Information

TSP was first formulated as a mathematical problem in 1930 and is one of the most intensively studied problems in optimization. Hassler Whitney and Merrill Flood at Princeton later investigated it. Later, in 1940"s the TSP was studied by statisticians in relation to agricultural application. Solution methods of TSP began to appear in papers in the mid-1950s; these papers used a variety of minor variations of the term "Traveling Salesman Problem." In the following decades, many researchers from mathematics, computer science, chemistry, physics, and other sciences studied the problem. Although TSP is easy to understand, it is very difficult to solve. Richard M. Karp showed in 1972 that the Hamiltonian cycle problem was NP-complete (non-deterministic polynomial time hard), which implies the NP-hardness of TSP. NP-hard, is a class of problems that are informally the hardest problems in NP which means no polynomial-time algorithm is known for solving TSP. This supplied a scientific explanation for the apparent computational difficulty of finding optimal tours.

Brief history of TSP milestones- Dantzig, Fulkerson, and Johnson published a description of a method for solving the TSP and illustrated the power of this method by solving an instance with 49 cities in 1954. It turned out that an optimal tour through the 42 cities used the edge joining Washington, D.C. to Boston; since the shortest route between these two cities passes through the seven removed cities, and this solution of the 42-city problem yields a solution of the 49-city problem. Held and Karp solved a TSP using 64 cities in 1971. Later in 1975, Camerini, Fratta, and Maffioli solved a TSP with 67 cities. In 1977, Grotschel solved a TSP using 120 cities in and around Germany and so on and so forth in later years. In 2004, TSP of visiting all 24,978 cities in Sweden was solved; a tour of length of approximately 72,500 kilometers was found and it was proven that no shorter tour exists.

TSP is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved. The TSP is represented in numerous transportation and logistics applications such as:

- Arranging routes for school buses to pick up children in a school district,
- Delivering meals to home-bound people,
- Scheduling stacker cranes in a warehouse,
- Planning truck routes to pick up parcel post and many others.
- Planning, logistics, and the manufacture of microchips.
- A classic example of the TSP is the scheduling of a machine to drill holes in a circuit board.

Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept city represents for example, customers, soldering points, or DNA fragments, and the concept distance, represents travelling times or cost, or a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows make the problem considerably harder. In the theory of computational complexity, the decision version of the TSP (where, given a length L, the task is to decide whether any tour is shorter than L) belongs to the class of NP-complete problems. Thus, it is likely that the worst case running time for any algorithm for the TSP increases exponentially with the number of cities.

III. Literature Survey

This section presents a literature survey done on algorithms used to solve TSP and detailed description of the ones that are GPU-friendly. For all optimization methods described to solve TSP, the method in question is described through insights gained from a survey of papers. The solutions below are divided into three main categories: (A) Swarm Intelligence metaheuristics (B) Population based metaheuristics and (C) Local search or trajectory based metaheuristics. The survey compares them on how various solutions were improved and scaled to gain speedup. Some of the algorithms in detail below are highly parallel where the problem is divided into procedures each running independently thus accounting for large problem sizes making these solutions GPU-friendly. The survey compares different solutions based on Quality, Scalability, Complexity and GPU-friendliness.

A. Swarm intelligence metaheuristics -

This is based on communication between many but relatively simple agents. One method in this for GPU implementation is ***Ant Colony Optimization (ACO).*** 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 max–min 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.

Below you can see the papers implementing ACO for TSP showing which steps of ACO are performed on the GPU in what fashion and how scalable the solution is i.e. change of processing power effect on speedup:

1. Bai et al. (2009) uses Multi colony algorithm MMAS (Max-Min Ant System) with GPU GeForce 8800 GTX achieving a maximum speed-up of 2.3. One colony per block assigned.
2. Li et al. (2209a) uses MMAS (Max-Min Ant System) with GPU GeForce 8600 GT and a maximum speed-up of 11
3. Wang et al. (2009) uses MMAS with GPU Quadro Fx 4500 and a speed of 1.1
4. You (2009) uses ACO (Ant Colony) with GPU Tesla C1060 achieving max speed of 21
5. Cecilia et al. (2011) uses ACO (Ant Colony) with GPU Tesla C2050 achieving max speed of 29
6. Delevacq et al. (2013) uses MMAS & multi-colony with GPU 2 * Tesla C2050 achieving max speed of 23.6
7. Uchida et al. (2012) uses Ant System with GPU GeForce 580 GTX and max speed 43.5

With the arrival of CUDA and OpenCL, programming the GPU became easier and consequently more papers studied ACO implementations on the GPU. In CUDA and OpenCL there is the basic concept of having a thread/workitem as basic computational element. Several of them are grouped together into blocks/workgroups.
We observe that for the ACO, the task most commonly executed on the GPU is tour construction. The papers of Cecilia et al. (2011) and Delévacq et al. (2013) indicate that the one-ant-per-block scheme seems to be superior to the one-ant-per-thread scheme.

Pros and Cons of ACO-
Pros:
- It has advantage of distributed computing.
- It is robust and also easy to accommodate with other algorithms.
- ACO algorithms have advantage over simulated annealing and Genetic Algorithms approaches of similar problems (such as TSP) when the graph may change dynamically, the ant colony algorithms can be run continuously and adapt to changes in real time.

Cons:
- Though ant colony algorithms can solve some optimization problems successfully, we cannot prove its convergence.
- It is prone to falling in the local optimal solution because the ACO updates the pheromone according to the current best path.

One of the ACS's is discussed below in detail.

SOLVING GEOMETRIC TSP WITH ANTS (2005)

Modified Ant Colony System with local optimization (2-opt, 3-opt) in multi-stages ->
Geometric TSP- vertices of the graph are points in the Euclidean plane and the weight of an edge is just the Euclidean distance between the two end points of the edge.
mACS (modified ant colony system algorithm) - An algorithm that is a combination of an ant system and an efficient local optimization algorithm(2-opt, 3-opt) is used with a preprocessing stage that allows us to deal with larger input instances efficiently. The algorithm provides optimal solutions within 0.3% of the optimum.
The algorithm consists of two main phases. In the first phase compacting given input graph into a smaller graph creates a good starting tour. A good starting tour allows mACS to find a better solution (TSP tour) and converge in much less time. The overall effect is that we can deal with larger input instances, producing good solutions in small amount of times. This TSP tour is then converted into a tour for the original input graph, and is locally optimized (2-opt, 3-opt). It is then used as the starting point for the second phase to create a better tour. In phase two, the tour found by mACS is returned as the solution to the input instance.
In mACS, ants explore more at the beginning and tend to exploit more toward the end of the algorithm, utilizing the information that have been accumulated. More exploitation may direct ants to use the edges of the global best tour more frequently. To avoid a local optimum, the algorithm perturbs the global best tour by erasing some memory from some percentage of randomly chosen edges of the tour. Additionally, when ants find the same tour over and over again or do not improve the global best tour after a certain number of iterations the algorithm encourages the exploration of edges not used frequently.

Pros:
- Positive feedback which accounts for rapid discovery of good solutions
- Distributed computation avoids premature convergence
- Greedy heuristic heuristics helps find acceptable solution in the early solution in the early stages of the search process
- Collective intelligence of a population of agents

Cons:
- Slower convergence than other heuristics
- Performs purely for TSP problems larger than 75 cities

Verdict: Experimental results show that this algorithm outperforms Ant Colony System and is comparable to Max-Min Ant System for Euclidean TSP instances. mACS is GPU-friendly in the sense that it allows for larger instances of problem efficiently using parallelism as it's a combination of AS and 2-opt,3-opt. The algorithm generates optimum solution to 0.3% of optimum thus getting good quality of results. The various speedups of ACO and its scalability are seen above. To overcome performance loss, we should have enough data to ensure a lot of parallelism in GPU.

B. Population-based metaheuristics -

Genetic Algorithm (GA) at a glance– A search technique to find approximate solutions of optimization over the resulting graph. It is an evolutionary algorithm i.e. a population of solutions evolves over time, yielding a sequence of generations. A new population is created from the old one using a process of reproduction and selection, where the former is often done by crossover and/or mutation and the latter decides which individuals form the next generation.

Below you can see the papers implementing GA or an immune evolutionary algorithm (IEA), which combines concepts from immune systems with evolutionary algorithms for TSP using CUDA:

1. Li et al. (2009b) uses IEA with operators PMX (partially mapped crossover), mutation and GPU GeForce 9600GT
2. Chen et al. (2011) uses GA with operators crossover, 2-opt mutation and GPU Tesla C2050
3. Fujimoto and Tsutsui (2011) uses GA and operators OX (Order crossover), 2-opt local search gene string move and GPU GeForce GTX 285
4. Zhao et al. (2011) uses IEA and operators Multi bit exchange and GPU GeForce GTS 250

We see that the choice of different operators, results in different optimal solutions.

The paper below discusses one implementation using Genetics Algorithm-

PERFORMANCE METRICS AND EVALUATION OF A PATH PLANNER BASED ON GENETIC ALGORITHMS

Genetics Algorithm to solve combinatorial path planning problem ->

Genetic Algorithm works well, both for the simple Subtour problem and the classic TSP. For the *Subtour* problem (for a given set of n targets, the agent has to find the shortest path (the Subtour) that visits k targets out of the n possible ones), an innovative Genetic Algorithm path planner is used, which finds the near-optimal strategy that allows the salesman to accomplish a *given* set of n interesting targets in the shortest amount of time. This application of GPU can be divided to similar procedures working on different data with not very severe dependencies among procedures. This approach is used to find solution to Travelling Salesman Problem in much less time. Although it might not find the best solution, It can find a near perfect solution for 100 city tour in less than a minute. Genetic Algorithm is a search technique used to find approximate solutions to combinatorial optimization problems. It is based on natural evolution and includes the survival of the fittest idea algorithm.

In a GA, a chromosome represents every possible solution, which is a sequence of values. The algorithm works with *population* of candidate solutions, evaluated by a predefined *cost function* and only the fittest surviving (evolution). The GA tries to minimize the cost of the chromosomes by repeating the process of combining and modifying them. The operations used to emulate the evolution process of a GA are *selection*, *crossover*, and *mutation*. The selection operator takes the responsibility of guiding the search of GA toward the high quality or even optimal solution. The crossover operator plays the role of exchanging the information between the individuals in the population while the mutation operator is used to generate a new offspring by randomly swapping genes and/or randomly changing a gene to another one not already present in the chromosome thus avoiding GA from falling into local optima.

Performance: Performance metrics of the algorithm is defined in terms of optimality of the solution and computational time. GA is one of the algorithms that sacrifices the optimality for a near-optimal solution obtained in shorter time. To quantify the speed of convergence with various genetic operators and the 2-opt method, the required number of generations for the convergence of the algorithm and the associated computational time are compared to conclude that the double cutting point crossover coupled with the mutation operator provides the highest speed of convergence, while a GA with the only single cutting point crossover needs more generation steps to reach close to the optimal solution.

→ A common approach for improving the TSP solutions is the coupling of the Genetic Algorithm with a heuristic boosting technique. The local search method is the well-known *2-opt method* for the TSP. This method as mentioned above replaces solutions with better ones from their neighborhood and provides a shorter path with no intersecting links.

Pros of GA

– Best solution using "fitness criteria"
- Capable of solving any optimization problem based on chromosome approach
- Capable to handle multiple solution search spaces and solve the given problem in such an environment. It supports multi-objective optimization.
- Intrinsically parallel
- Always an answer; answer gets better with time
- Better quality of solutions and cost as well as solution times
- It is more useful and efficient when search space is large, complex and poorly known or no mathematical analysis is available.
- The GA is well suited to and has been extensively applied to solve complex design optimization problems because it can handle both discrete and continuous variables, and nonlinear objective functions without requiring gradient information.

Cons of GA

- No optimal solution; approximation of solution is reached but not an optimal solution
- Population for evolution should be moderate
- Crossover rate should be 80% -95%
- Method of selection and writing of fitness function should be appropriate
- When fitness function is not properly defined, GA may converge towards local optima.
- Operation on dynamic sets is difficult.
- GA is not appropriate choice for constraint based optimization problem.

Verdict- Evolutionary algorithms provide clear parallelism making it GPU-friendly. The computation of offspring can be performed with at most two individuals (the parents). Moreover, the crossover operators might be parallelizable. Quality: Either way, enough individuals are needed to fully saturate the GPU, but at the same time all of them have to make a contribution to increasing the solution quality. Apparent parallelism has led to the two parallelization schemes of assigning one individual to one thread (coarse grained parallelism) (Chen et al. 2011) and one individual to one block (fine grained parallelism) (Li et al. 2009b; Fujimoto and Tsutsui 2011). The scheme chosen obviously influences the efficiency and quality of the GPU implementation. Increasing Problem size (complexity): On one hand a minimum number of individuals is needed to fully saturate all of the computational units of the GPU, especially with the one-individual-per-thread scheme. On the other hand, from an optimization point of view, it might not increase the quality of the algorithm to have a huge population size. Scalability: The speedup compared with unknown CPU implementations is highest for Li et al. (2009b).

Combination of GA and Ant –

There are some shortcomings if only one of them is used to solve TSP. We can overcome the shortcomings if GA and ACS are combined to solve TSP and get faster convergent speed and more accurate results compared with only using ACS or GA. ACA and GA are all bionic probabilistic search algorithms which combine with distributed computing, tend to parallel computing and are more robust. ACA simulates group behaviors between communities and environment made up with simple individual, which may cause unpredictable group behavior, but GA simulate genetic evolutional process. They have common grounds on carrying out Traveling Salesman Problem (TSP):

1) They are all prone to premature convergence, so as to get into local optimum value
2) They have not special requirements to searching space and their practical ranges are more widespread.

Difference: Ant colony algorithm converges to the optimal path through the accumulation and update of information, but the lack of pheromone in initial stages leads to slower speed of convergence.
Genetic Algorithms have rapid global searching capability, but the feedback of information in the system has not been utilized, sometimes leading to do nothing redundant iteration and inefficient solution. Therefore, if the size of the cities are more than 30, genetic algorithms' searching capability will gradually decline, and when the number of the cities are too big, it cannot obtain the optimal solution infinite iteration because iterative times are too long and unbearable. Here ant colony algorithm is better than genetic algorithm, and it can reach the optimum value. When the size of cities is too big, ant colony algorithm may appear stagnation. Thereby it cannot obtain optimum value.
Results:
Therefore, the use of randomly search and rapid of genetic algorithms, and pros-feedback mechanism and high efficiency characteristic of ant colony algorithm - Better solutions are produced and pheromones are left behind the paths of its, pheromones in other paths are not changed. And then, ants perform operations of crossover and mutation in accordance with GA after each ant completes one traveling in the light of ACA.

C. Local search and trajectory-based metaheuristics

Given a current solution, the idea in ***Local Search*** is to generate a set of solutions—the neighborhood—by applying an operator that modifies the current solution. The best solution is selected, and the procedure continues until there is no improving neighbor, i.e., the current solution is a local optimum. Scalability: Many papers implement local search variations on the GPU, reporting speedups of one order of magnitude when compared with a CPU implementation of the same algorithm. Profiling and fine-tuning the GPU implementation may ensure good utilization of the GPU. Schulz (2013) reports a speedup of up to one order of magnitude compared with a naive GPU implementation. To fully saturate the GPU, the neighborhood size is critical; it must be large enough (Schulz 2013).

The following GPU papers present local search and the speedup achieved in each:

1. Luong et al. (2011b) uses Local search with 2-exchange (swap) in CUDA and GPU amongst others Tesla M2050 achieving max speed-up of 19.9
2. O-Neil et al. uses Multi-Start Local Search with 2-opt in CUDA: multiple-ls-per-thread, load balancing and GPU Tesla C2050 and max speed-up of 61.9
3. Burke and Rise (2012) uses Iterated Local Search with VND (Variable neighborhood descent): 2-opt + relocate in CUDA: one-move-per-thread, applies several independent moves at once and GPU

GeForce GTX 280 achieving max speed-up of 70 *7.5

4. Rocki and Suda (2012) uses Local Search with 2-opt, 3-opt in CUDA: several-moves-per-thread and GPU amongst others Geforce GTX 680 and max speed-up of 27

Pros: If one thread on the GPU is responsible for the evaluation of one or several moves, a mapping between moves and threads can be provided. The advantage of the mapping approach is that there is no need for copying any information to the GPU on each iteration. The pre-generated mapping only needs to be copied to the GPU once before the LS process starts.

Cons: *Increasing Problem size(complexity) and scalability:* There are efficiency aspects and limitations of local search on GPU. In CUDA it is not possible to synchronize between blocks inside a kernel. The efficiency of a kernel is obviously important for the overall speed of the computation. For very large neighborhoods, a lot of computational effort is wasted and might not fit into GPU memory and is guaranteed to get stuck. Multi-start however, provides parallelism where one local search instance is independent of other. So there are 2 approaches- GPU-based parallel neighborhood evaluation of different local searches performed sequentially or local searches run in parallel on GPU. But both have their drawbacks too. O'Neil et al.(2011) uses load balancing as a solution.

Two papers below are discussed in detail where one presents Local Search compared to the other presenting Multi-search technique. We see multi-search to be better.

1. ACCELERATING 2-OPT AND 3-OPT LOCAL SEARCH USING GPU IN THE TSP (2012)

2-opt and 3-opt Local Search Algorithm ->

In the symmetric TSP, the distance between two cities is the same in each opposite direction, forming an undirected graph. The most direct solution for such a TSP problem would be to calculate the number of different tours through N cities.

One method for solving the problem is to use the 2-opt algorithm which removes two edges from the tour, and reconnects the two paths created so that the tour remains valid and repeat only if the new tour would be shorter. Continuing removing and reconnecting the tour until no 2-opt improvements can be found leads to an optimal route (local minimum) in polynomial time (O (N^2) complexity). Reconnecting and reversing order of sub-tour improve tour.

The 3-opt algorithm works in a similar fashion, but instead of removing two edges, three are reconnected. A 3-opt exchange provides better solutions, but it is significantly slower (O (N^3) complexity).

Scalability: It is a very important local search technique and using GPU to parallelize the search speeds up optimization by greatly decreasing the time needed to find the best edges to be swapped in a route approximately by 100 times in case of 2-opt compared to a sequential CPU code and more than 220-fold speedup can be observed in case of 3-opt search achieving more than 430 GFLOPS on a single Tesla C2075 GPU.

Iterative Local Search (ILS) uses 2-opt search with random restarts to escape from local minima achieving a global solution if the search time is infinite, assuming sufficient randomness. At least 90% of the time during an Iterative Local Search is spent on the 2-opt itself and that number increases with the problem size growing.

2. An efficient GPU implementation of a multi-start TSP solver for large problem instances (2012)

Modified iterative hill climbing algorithm -> A constructive multi-search algorithm that uses parallel GPU implementation (over 170 GFLOPS on single TESLA C2050 card running thousands of independent threads) to solve large instances of the TS problem but 2 times slower (90% longer) than the original GPU implementation of IHC. Through this approach an approximate solution to the problem of size up to 6000 cities can be achieved efficiently. The way IHC works is that an initial solution is improved using heuristic techniques until a locally optimal solution is reached. This modified IHC is built on the original GPU implementation proposed in paper by O'Neil et al. Their approach pre-calculates distances between cities and stores them in shared memory of GPU. The algorithm ran on one GPU chip which was 62 times faster than the corresponding serial CPU code, 7.8 times faster than an 8-core Xeon CPU chip, and about as fast as 256 CPU cores (32 CPU chips) running an equally optimized pthreads implementation. But this approach is limited to GPU memory size hence allowing the largest problem size to be 110 cities. To make their approach more universal and generalized, the modified version of IHC uses multiple GPU's to parallelize even more and exploits its power to calculate distances between cities 'on-the-fly'. The comparison between the original and modified shows that the modified IHC started with using global memory rather than shared memory of GPU, but achieved high latency thus then moves on to using coordinates in shared memory i.e. not to calculate the data by CPU, but just to read the coordinates and transfer them to the GPU to the fast on-chip shared memory for low-latency access. Finally using less precise sqrt function and fast math compilation option leads to usage of faster equivalents of mathematical functions on the GPU and shortened the time.

Why this is used as GPU application is because modified IHC can involve consuming large amount of data. It is believed that IHC approach may be better suited for GPU-based acceleration than the related ant colony and genetic algorithm-based TSP solvers that are available for GPUs.

Using a GPU to solve-

For Parallelism, more than 10,000 threads are needed. Sets of 32 threads need to have good access to memory and need to follow the same control flow. There are atleast O (n^2) operations on O(n) data. Assuming 100-city problems & 100,000 climbers where climbers are independent and can be run in parallel.

Scalability and problem size: But Using 2 types of new performance indicators (i) FLOPS (Floating Operation per Second) and (ii) the number of 2-opt exchanges per second (EX/s), it is observed that the GPU is very fast in terms of FLOPS, and that justifies its usage to calculate the distances rather than just reading them from the memory. As the problem size increased, the performance was slightly better as well (from 109.4 GFLOPS to 117.1 GFLOPS in case of 10000 climbers and from 167.7 GFLOPS to 172.8 GFLOPS in case of 100000 climbers).

A scalable implementation to get significant speedup for problems larger than 1000 cities can be achieved on the TSUBAME 2.0 supercomputer using hundreds of available GPUs. Parallelizing the 2-opt search itself on GPU can do this.

Pros

- Plenty of data parallelism
- For finding best opt-2 move, code can be optimized
- Using doubly nested loop; so computes difference in tour length, not absolute length
- Minimizes memory accesses; caches rest of data in registers and requires only 6 clock cycles per move on a Xeon CPU core
- Local minimum compared to best solution so far; best solution updates if needed, otherwise tour is discarded
- Random tours generated in parallel on GPU; minimizes data transfer to GPU
- 2D distance matrix resident in shared memory; ensures hits in software-controlled fast data cache
- Tours are coped to local memory in chunks of 1024; enables accessing them with coalesced loads & stores

Cons

- Potential load imbalance
- Different number of steps required to reach local minimum

Verdict: We see that a parallel GPU Version of TSP by O'Neil using Iterative Hill Climbing finds optimal solutions for four out of five 100-city TSPLIB inputs and that this implementation is highly parallel making it GPU-friendly. Speedup over Serial – Pthreads code scales well up to 32 threads (4 CPU's). GPU is stable while CPU performance fluctuates. We see that TSP GPU algorithm using IHC is a highly optimized implementation for GPU's. It evaluates 20 billion tour modifications per second on a single GPU (as fast as 32 8-core Xeons) and produces high-quality results. This may be better suited for GPU than Ant Colony Optimization and Genetic Algorithms. To account for large problem sizes we can use the modified IHC.

GPU-friendly algorithms discussed above use parallel implementations. Each core can be assigned a subset of problem. Each application can be divided to similar procedures (working on different data) independent of each other.

IV. Solution

This section provides an adequate background of the specific solution implemented to solve TSP and the functionality of these implementations are demonstrated. This project created an implementation for solving the Travelling Salesman Problem in CUDA through the use of Genetic Algorithm (GA) and Ant Colony Optimization (ACO). Both these algorithms are highly parallel- where the problem is divided into procedures each running independently thus accounting for large problem sizes making these solutions GPU-friendly. Looking at various advantages of GA and ACO in the above section these GPU-friendly algorithms were picked amongst others based on the following factors-

- Both these parallel algorithms work well for changing problem size and its effect on the complexity overall. They are efficient in large complex problem sizes.
- Time the programs take to run – results are achieved faster
- Speedup – CPU/GPU
- Better quality of results i.e. near optimal solutions
- They are robust and easy to accommodate with other algorithms.

Both these solutions are then compared in the results section based on the same factors, GPU utilization and how results change by changing number of cities, ants, maximum tours, generations and population.

1. Genetic Algorithms

Genetic Algorithm was used as one method to find a solution to the TSP. A Genetic Algorithm is one of the oldest and most successful optimization technique based on natural evolution. Again, a GA is a search heuristic that utilizes the process of natural selection to arrive at a desirable solution. This uses survival of the fittest approach for selecting the best (fittest) solution from the available solutions. GAs are designed to maximize a fitness function. In the TSP it is desired to minimize the distance; thus, the fitness function was set to be 1/ distance. Calculating the distance is costly as it involves computing a square root. Since distances can be compared accurately without having to take the square root (using the squared Euclidean distance), this operator was removed, resulting in the fitness function shown below, where cities is a list of objects containing the x and y coordinates for each city in the world.

$$\frac{1}{\sum_{i=1}^{numCities-1} [cities[i+x].x - cities[i].x)^2 + (cities[i+1].y - cities[i].y^2]}$$

TSP consists of finding the minimum distance between fixed cities; thus, the distance is a function of the order the cities are traveled through. For this reason, a permutation-encoding scheme was selected. In a permutation-encoding scheme, the only adjustable feature is the order of the objects. A GA works on a provided population of individuals. For this context, each individual in the population will be an instance of the desired world, where a world consists of a list of randomly arranged cities. Note that the same cities must be used for each world instance, ensuring that the salesman travels through the same set of cities, each time.

Generations

A GA operates on generations, where each generation produces a new population from the previous population, and a dominant individual emerges. The stopping condition for this project was set to be a user-defined, fixed number of generations. The final result is obtained by finding the best route (largest fitness function) across all of the generations. Within a generation, a GA must make a new population, utilizing the previous population. Creating a new population involves performing a selection, crossover, and mutation step.

The process of selection, crossover, and mutation is iteratively performed until a new population is created that is the same size as the original population. It should be noted that there exists a random probability of both crossover occurring and a random probability of mutation occurring. If crossover does not occur, the first parent is utilized as the new child. With that new child, the algorithm continues, with mutation. If mutation does not occur on that child, it is simply inserted into the new population as is, resulting in a direct copy of one of the parents.

Working of GA

Step 1: Selection
In selection, two parents are chosen from the population. There are a number of different selection schemes, all having the basic goal of selecting the individuals with the top fitness functions. This process simulates the idea of "survival of the fittest". The selection scheme utilized in this project is known as roulette wheel selection. This selection scheme consists of three steps. In the first step, the sum of the fitness function across all individuals is calculated. The second step is used to calculate the individual probabilities, which is simply the individual's fitness divided by the sum of the fitness of all individuals up through the current individual. The third step is used to select the two parents. A random probability is generated and the population is iterated. If the individual's probability is greater than the random probability, it is selected as a parent. This step is repeated until both parents are selected.

Step 2: Crossover
After the parents are selected, they are mated to create children. This step is known as crossover, as traits from each parent are used to create one or more children. In this work, the single point crossover method was utilized, where only one child is created per set of parents. This involves randomly selecting a crossover point, copying all elements up through that point, from the first parent, into a new child, and then adding the rest of the elements from the second parent. This process is illustrated below. It is important to note that the set of cities cannot be changed; therefore, when cities are added from the second parent, only cities that are not currently in the child will be added. Cities are added sequentially from the second parent, beginning with the first city, to ensure that the order is maintained as best as possible.

Step 3: Mutation
The final step of a GA is to mutate the newly created child. This simulates the birth of a child with features that are unique to it. The mutation scheme utilized in this work was the order changing permutation. In this method, two unique elements are randomly chosen and swapped. Repeat selection, crossover and mutation operations to produce more new solutions until the population size of the new generation is the same as that of the old one. The iteration then starts from the new population. Since better solutions have a larger probability to be selected for crossover and the new solutions produced carry the features of their parents, it is hoped that the new generation will be better than the old one. The procedure continues until the number of generations is reached to n or the solution quality cannot be easily improved.

Implementation

Pseudo Code of GA

```
begin procedure GA
   generate populations and fitness function
   evaluate population
   while(termination criteria not met)
   {
     while (best solution not met)
   {
     crossover
     mutation
     evaluate
   }
}
post-process results and output
end GA procedure
```

Sequential implementation

Some changes were made to improve performance of the code. The first two steps of selection process were instead added into the evaluation process thus removing an entire loop in code. Also, only one child was produced per set of parents to ensure that iterations of size of number of individuals would be required to create each population. In order for the deterministic path to be taken, a random seed was used to pre-generate random numbers for timing and repeatability process. Memory management was tricky as increasing the population size or number of cities results in a large increase in memory utilization.

Parallel implementation

It was found that the largest speedup could be obtained only by exploiting parallelization within a generation. So all the operations of creating a new population, calculating fitness function and selecting dominant leader may be done in parallel. Memory management was the biggest challenge while mapping sequential code to CUDA as these memory operations had to be manually rewritten. Global memory was found to be best suited where a copy of population created in CPU is placed. All other operation takes place in device memory. For a direct-to-direct comparison the random numbers weren't generated in GPU and large amount of memory was thus needed. The routes were traced and dominant individual from each generation was copied back to CPU. Since only a single transfer is made, the data transfer time is negligible.
In the code there are six kernels and two device functions where three kernels were used to evaluate fitness function. Out of the three, the first one computes actual fitness function and each thread calculates sum of squared Euclidean distances for its cities. The second one calculates partial fitness sum to determine partial probabilities during selection step. The third kernel calculates individual fitness probability. To select the dominant leader, maximum fitness must be computed for which a kernel and one thread. Another kernel is used to select the parents. Here each thread selected one parent thus double 'population size' threads were required but the thread's work is done as soon as a parent is found. The last kernel was used to create children. Crossover and mutation is done via device functions. After each generation, the old and new populations are swapped. In crossover, both old and new populations have to be iterated. Mutation is a simple pointer swap operation.
It was seen that there was a substantial improvement in performance of GPU over CPU.

2. Ant Colony Optimization (ACO)

Implemented a sequential and parallelized Ant Colony Optimization algorithm applied to TSP. ACO is a probabilistic, optimal path algorithm inspired by the natural behavior of ants scouting for food. It is a population-based general search technique for the solution of difficult combinatorial optimization problems. Individual agents (ants) randomly travel along edges of a graph, searching for the target vertex. When an ant arrives, it returns to the colony (source), leaving *pheromone* markers for other ants to see. These pheromones increase the probability of a later ant following that path. Over time, however, pheromones decay/evaporate. As more ants follow the path and drop more pheromones, shorter paths are encouraged and the algorithm converges to a nearly optimal solution. I used CUDA to implement a speedup on the GPU over the baseline sequential CPU algorithm.

The algorithm takes in a complete graph on *n* vertices as input generated by a Map Generator that is open source and outputs an approximate "best" tour and tour length based on the ACO algorithm. The most computationally expensive part of the algorithm in the sequential implementation is the simulation of the ants. In the CUDA programming model the computation required for choosing the next cities contains much arithmetic that can be parallelized by running across multiple blocks and threads. The algorithm consists of two steps, which can be parallelized: tour construction (in which each ant picks the next edge to travel), and pheromone update (in which pheromones are added to successful paths). The pheromone values for each edge in the graph must be updated for every ant in the simulation. This implies the need for synchronization between ants that want to update the same edge.

Implementation

Pseudo Code of ACO

```
begin procedure ACO
    generate pheromone trails and other parameters
    while(termination criteria not meet)
    {
        construct solutions
        update pheromone Trails
    }
    post-process results and output
end ACO procedure
```

Initially ants start their search roams randomly. An ant selects the next node to be visited by probabilistic equation. To find the probability of going to node j when ant k is on node I, the parameters involved are- number of ants, 'alpha' which is the local pheromone coefficient that controls the amount of contribution pheromone plays in a components probability of selection and is set to 1 and 'beta' which is the heuristic coefficient which controls the amount of contribution problem-specific heuristic information plays in a components probability of selection and is set to 5. The arcs which are used by the most ants and which is the shortest, receives more pheromone and will be used by the ants in future. Another main function is pheromone update which consists of pheromone deposit and pheromone evaporation. Pheromone values are updated each time an ant travels from one node to another. A first Pheromone value on each arc is decreased by constant factor, which is known as pheromone evaporation. Then some amount of pheromone is added to each node which is being traversed by each ant, is known as pheromone deposit. Each ant drops some amount of pheromone on each node, which is known as pheromone deposit. The evaporation rate is set to 0.5. These weights or "pheromones" on the edge are updated on each iteration of the algorithm based on the number of ants that have traversed the edge and the length of those ants' resulting tours (more ants, shorter paths leads to higher pheromone values, indicating a more favorable edge).

I mapped each ant to a thread and distributed the cities in the graph to the threads. During the tour construction phase, each thread would compute in parallel the likelihood that the ant would visit any city in the graph, and randomly select one based on the probabilities. During the pheromone update step I mapped each ant to a thread and atomically updated the edges. In the sequential version of the code all ants are simulated before stepping into the next city.

Challenge faced-

Updating Pheromones

Updating the pheromones efficiently in parallel is a problem as well. Because ants could leave pheromones along the same edge of a graph at a time, we need to worry about synchronizing access to the pheromones' storage in CUDA, in addition to high costs of loads and stores to global memory. Basic atomic sets and gets might not cut it for performance.

Solved by – used cudaThreadSynchronize to make sure all threads are completed before stepping to next tour.

V. EXPERIMENTAL SETUP

To obtain the results the code was run on CIMS using cuda2 with the following device specifications:

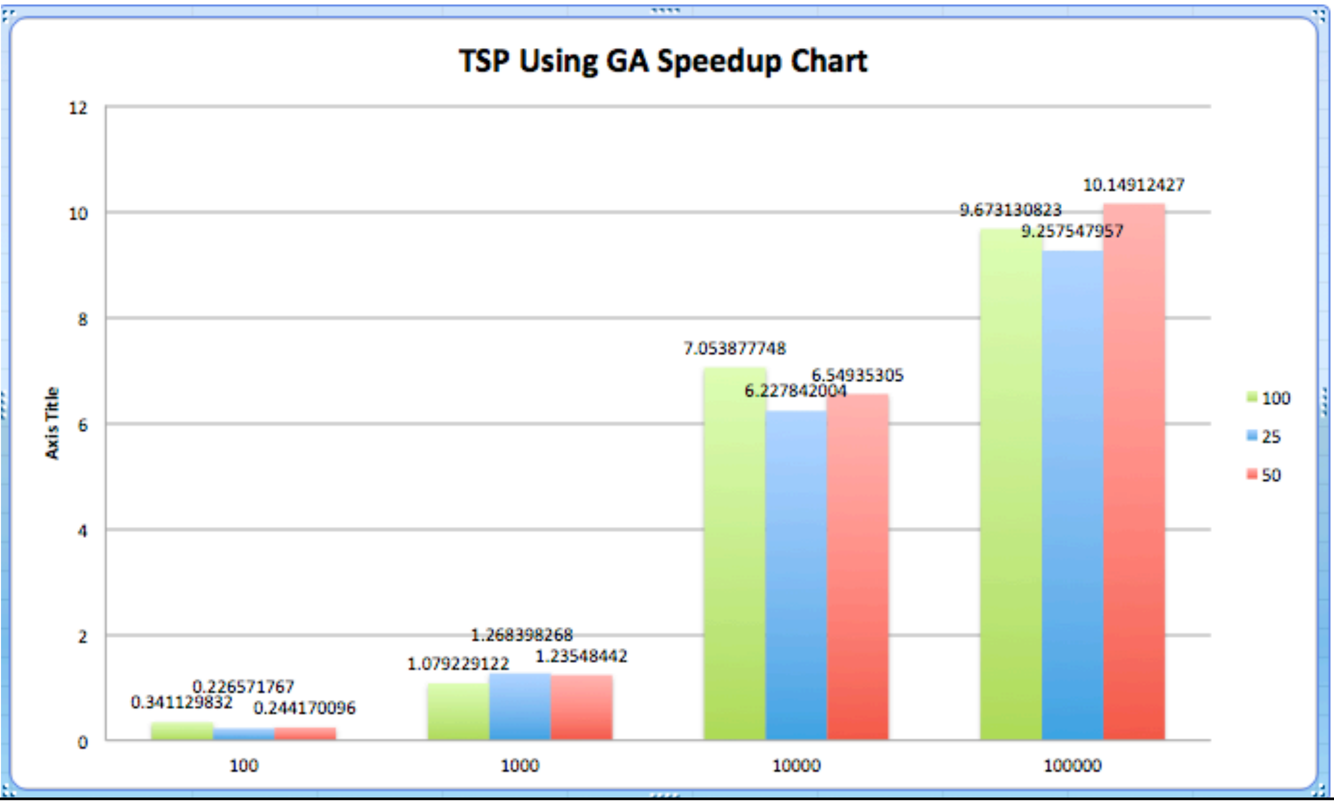
name: GeForce GTX TITAN Z
Compute capability 3.5
total global memory(KB): 6227968
shared mem per block: 49152
regs per block: 65536
warp size: 32
max threads per block: 1024
max thread dim z:1024 y:1024 x:64
max grid size z:2147483647 y:65535 x:65535
clock rate(KHz):
total constant memory (bytes): 65536
multiprocessor count 15
integrated: 0
async engine count: 1
memory bus width: 384
memory clock rate (KHz): 3505000
L2 cache size (bytes): 1572864
max threads per SM: 2048

VI. EXPERIMENTAL RESULTS & ANALYSIS

Both techniques (Genetic Algorithm and Ant Colony Optimization) are used to solve travelling salesman problem with high acceptable performance. Results obtained from each algorithm are shown below and they are compared.

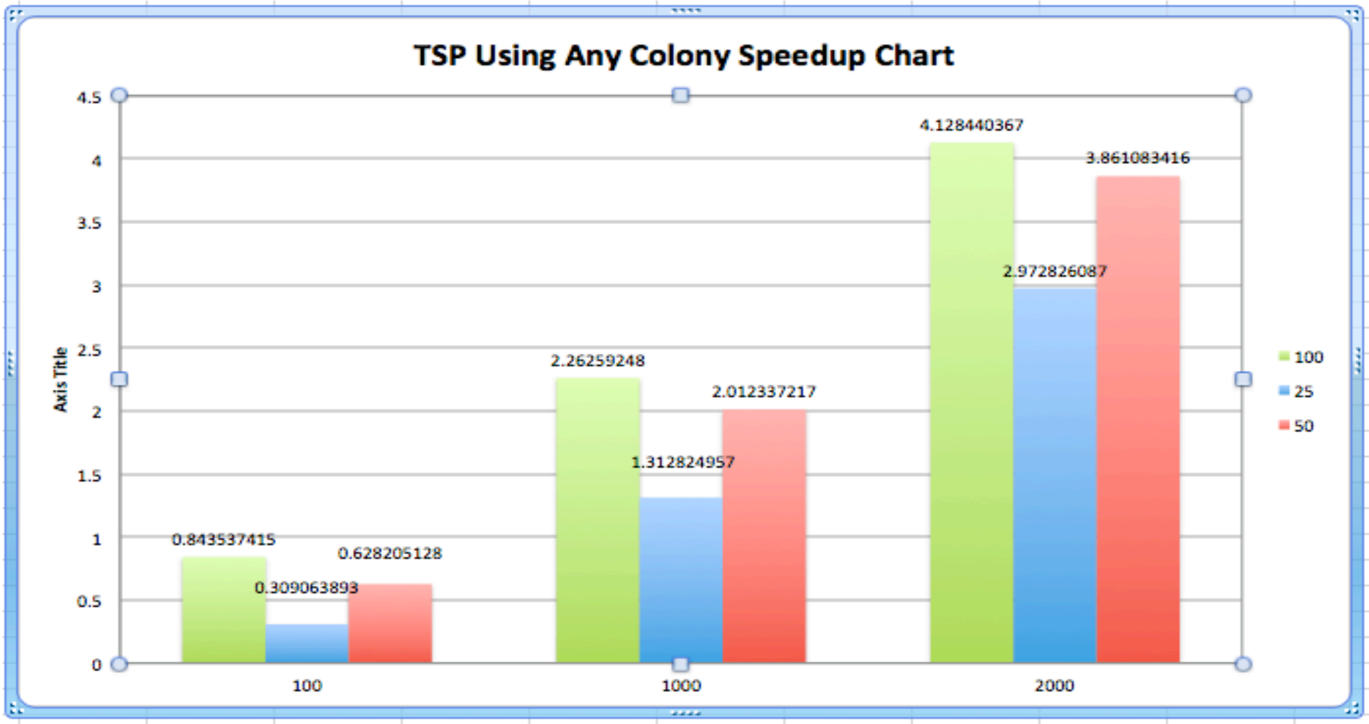
No. of Cities	Population Size	Max Generation	CPU Run Time(ms)	GPU Run Time(ms)	SpeedUp (CPU/GPU)
25	100	1000	382	1686	0.226571767
25	1000	1000	3809	3003	1.268398268
25	10000	100	12929	2076	6.227842004
25	100000	10	110998	11990	9.257547957
50	100	1000	712	2916	0.244170096
50	1000	1000	7256	5873	1.23548442
50	10000	100	17716	2705	6.54935305
50	100000	10	121688	11990	10.14912427
100	100	1000	1721	5045	0.341129832
100	1000	1000	15120	14010	1.079229122
100	10000	100	29196	4139	7.053877748
100	100000	10	127695	13201	9.673130823

Runtime of TSP Solver implemented using Genetic Algorithm



No of Cities	Ants	Max Tours	CPU Run Time(ms)	GPU Run Time(ms)	SpeedUp (CPU/GPU)
25	100	50	208	673	0.309063893
25	1000	50	1515	1154	1.312824957
25	2000	50	3282	1104	2.972826087
50	100	50	637	1014	0.628205128
50	1000	50	5872	2918	2.012337217
50	2000	50	11618	3009	3.861083416
100	100	50	2232	2646	0.843537415
100	1000	50	22325	9867	2.26259248
100	2000	50	45000	10900	4.128440367

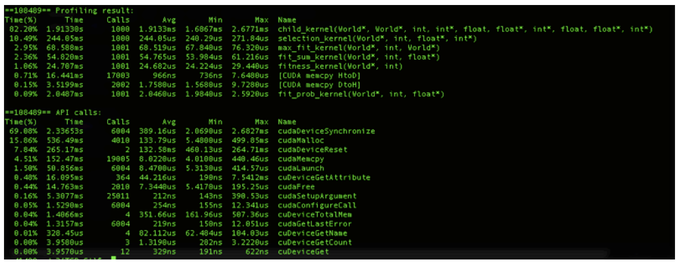
Runtime of TSP Solver implemented using Ant Colony Optimization



Experimental Results of TSP using GA

The parallel implementation of TSP using GA was run over a data consisting of variable number of cities, population and generations. The city map is generated using random numbers, using the seed parameter given in the starting of the code. It was made sure that the city randomly generated was same for CPU and GPU to get run time over the same data. And the following things were observed:

- As the population size increases the speed up also increases, and this was expected as the parallelization was exploited at the population level making the number of time the code is executed dependent on the population. Thus, as the population increased the speedup also increased as we were able to leverage more of the parallel computation provided by the GPU
- As the number of cities increases the speedup decreases, the reason behind this happening is that with the increase in the number of cities more amount of computation is required, which in result decreases the performance gains
- As the number of cities increase (with same amount of population) the speedup remains almost constant , which is unexpected as more cities means more amount of calculations. Taking a closer look using ‘nvprof’, we see that even though the CPU overhead increases, the execution time of GPU remains almost same due to the population size being same and high parallelization.
- A speedup as high as 11x was observed on a dataset containing 50 cities, and a high population size, where the code was running on the Specs mentioned above



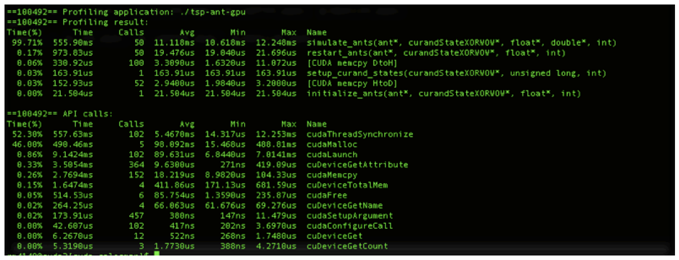
```
==108489== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 82.20%  1.91330s      1000  1.9133ms  1.6867ms  2.6771ms  child_kernel(World*, World*, int, int*, float, float*, int*, float, float*, int*)
 10.49%  244.05ms      1000  244.05us  240.29us  271.84us  selection_kernel(World*, int, float*, int*)
  2.95%  68.588ms      1001  68.519us  67.840us  76.320us  max_fit_kernel(World*, int, World*)
  2.36%  54.820ms      1001  54.765us  53.984us  61.216us  fit_sum_kernel(World*, int, float*)
  1.06%  24.707ms      1001  24.682us  24.224us  29.440us  fitness_kernel(World*, int)
  0.71%  16.441ms     17003     966ns     736ns  7.6480us  [CUDA memcpy HtoD]
  0.15%  3.5199ms      2002  1.7580us  1.5680us  9.7280us  [CUDA memcpy DtoH]
  0.09%  2.0487ms      1001  2.0460us  1.9840us  2.5920us  fit_prob_kernel(World*, int, float*)

==108489== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 69.08%  2.33653s      6004  389.16us  2.0690us  2.6827ms  cudaDeviceSynchronize
 15.86%  536.49ms      4010  133.79us  5.4800us  499.85ms  cudaMalloc
  7.84%  265.17ms         2  132.58ms  460.13us  264.71ms  cudaDeviceReset
  4.51%  152.47ms     19005  8.0220us  4.0100us  440.46us  cudaMemcpy
  1.50%  50.856ms      6004  8.4700us  5.3130us  414.57us  cudaLaunch
  0.48%  16.095ms       364  44.216us     190ns  7.5412ms  cuDeviceGetAttribute
  0.44%  14.763ms      2010  7.3440us  5.4170us  195.25us  cudaFree
  0.16%  5.3077ms     25011     212ns     143ns  390.53us  cudaSetupArgument
  0.05%  1.5290ms      6004     254ns     155ns  12.341us  cudaConfigureCall
  0.04%  1.4066ms         4  351.66us  161.96us  507.36us  cuDeviceTotalMem
  0.04%  1.3157ms      6004     219ns     150ns  12.051us  cudaGetLastError
  0.01%  328.45us         4  82.112us  62.484us  104.03us  cuDeviceGetName
  0.00%  3.9580us         3  1.3190us     282ns  3.2220us  cuDeviceGetCount
  0.00%  3.9570us        12     329ns     191ns     622ns  cuDeviceGet
```

nvprof on the ACO implementation with 50 cities and 1000 population size and max generations set to 1000

Experimental Results of TSP using ACO

The parallel implementation of TSP using ACO was run over a vast data consisting of different number of cities, different ants and fixed number of tours. Even though ant colony algorithm requires a low number (~150-200) of ants. We ran the program on a higher number of ants as well to see the results of the same as the parallelization was done around the number of ants, where each kernel thread simulates one ant. The following thing were observed:

- As the number of ants increase, a higher speedup was achieved, which was expected as the major amount of parallelization was done around the amount of ants, this increasing the number of ants gave a better speedup.
- As the number of cities increase, the speedup also increases, this happens mainly due to the fact that even though the overhead CPU time does increase, the time taken by the GPU doesn’t increases much, as it’s majorly parallelized around the number of ants. Giving us a better speedup with increase in number of cities
- Upon profiling the algorithm using ‘nvprof’, we found that we spend 90-95% of time performing tour construction and simulating the ants over it and around 5% of time updating the pheromones. Our speedup was limited by a lot of global memory accesses and the computation required to build each ant's tour.



```
==100492== Profiling application: ./tsp-ant-gpu
==100492== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
 99.71%  555.90ms        50  11.118ms  10.618ms  12.248ms  simulate_ants(ant*, curandStateXORWOW*, float*, double*, int)
  0.17%  973.83us        50  19.476us  19.040us  21.696us  restart_ants(ant*, curandStateXORWOW*, float*, int)
  0.06%  330.92us       100  3.3090us  1.6320us  11.072us  [CUDA memcpy DtoH]
  0.03%  163.91us         1  163.91us  163.91us  163.91us  setup_curand_states(curandStateXORWOW*, unsigned long, int)
  0.03%  152.93us        52  2.9400us  1.9840us  3.2000us  [CUDA memcpy HtoD]
  0.00%  21.504us         1  21.504us  21.504us  21.504us  initialize_ants(ant*, curandStateXORWOW*, float*, int)

==100492== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 52.30%  557.63ms       102  5.4670ms  14.317us  12.253ms  cudaThreadSynchronize
 46.00%  490.46ms         5  98.092ms  15.468us  488.81ms  cudaMalloc
  0.86%  9.1424ms       102  89.631us  6.8440us  7.0141ms  cudaLaunch
  0.33%  3.5054ms       364  9.6300us     271ns  419.09us  cuDeviceGetAttribute
  0.26%  2.7694ms       152  18.219us  8.9820us  104.33us  cudaMemcpy
  0.15%  1.6474ms         4  411.86us  171.13us  681.59us  cuDeviceTotalMem
  0.05%  514.53us         6  85.754us  1.3590us  235.87us  cudaFree
  0.02%  264.25us         4  66.063us  61.676us  69.276us  cuDeviceGetName
  0.02%  173.91us       457     380ns     147ns  11.479us  cudaSetupArgument
  0.00%  42.607us       102     417ns     202ns  3.6970us  cudaConfigureCall
  0.00%  6.2670us        12     522ns     268ns  1.7480us  cuDeviceGet
  0.00%  5.3190us         3  1.7730us     388ns  4.2710us  cuDeviceGetCount
```

nvprof on the ACO implementation with 50 cities and 1000 ants and max tours set to 50

Comparison

Here I have compared both the implementations of TSP using ACO and GA based on different factors:

- In general, ACO takes a much lesser time to reach the near optimal result when compared to GA with same *“problem size”* consisting of same amount of cities. This is majorly seen as the in ACO all the ants are working in parallel to find an optimal solution at each and every step, whereas in GA all the generations are calculated and the generation will best results is considered an optimal result.
- Even though ACO has a better run time in some cases in terms of *“GPU Runtime”*, but we can’t make a comparison on both of them based on the runtime as the computations are very different and depends on very different things (population size in GA and number of ants in ACO , which are not related to each other in any way)
- In terms of *“Speedup”* i.e. CPU/GPU, GA has a better Speedup when compared to the CPU version of the same, as the amount of work depends on the population size thus with data sets which have a higher population size, more parallelization is possible leading to a better speedup, but this doesn’t change the fact that ACO is still able to find much faster results when compared
- When we look at both these implementations while keeping *“GPU Utilization”* in mind, as we can see

above in the nvprof profiling, the number of kernel calls in ACO are much less compared to GA, since the kernel call in ACO are directly dependent on the number of ants whereas in GA they are depended on the number of generations to be calculated before selecting the optimal result. Also, in ACO, ~95-97% of work is done by there kernel and everything else is either copying of data or CPU functions. Whereas in GA its only ~90-95% of GPU usage. Based on this observation, ACO seems to be a more GPU friendly algorithm.

VII. CONCLUSION

This project presents a comparative view of most widely used optimization algorithm techniques namely ACO and GA to solve TSP. ACO is the process used by ants to forage food source. They use pheromone trail deposition/evaporation technique to map their way. GA is an optimization technique, inspired by the law of biological reproduction and survival of fittest theory, where mutation and crossover operations used to find by the local optimal solution.

Section 6 shows the experimental results obtained so to generalize even though GA is better in terms of speedup but ACO gives near optimum results faster making ACO better than GA. Both techniques have immense potential and scope of application ranging from engineering to software engineering, and real world optimization problems. These techniques need to be further explored to find their suitability to certain applications. Also, there is a need to combine two or more techniques so that they complement each other and nullify their respective limitations.

ACKNOWLEDGMENT

I'd like to thank Professor Z. for giving me the opportunity to explore this topic and the new technologies and equipping us with proper skillset to use these technologies.

REFERENCES

[1] Lawler E, Lenstra JK, Rinnooy Kan AHG, Shmoys DB. The travelling Salesman problem. New York: John Wiley & Sons; 1985.

[2] Dorigo M, Stützle T. Ant Colony optimization. Cambridge, MA: MIT Press; 2004.

[3] Cecilia et al. Parallelization Strategies for Ant Colony Optimisation on GPUs. NIDISC 2011.

[4] N. Sureja, B. Chawda, "Random Travelling Salesman problem using Genetic Algorithms," IFRSA's International Journal Of Computing,Vol. 2, issue 2, April 2012.

[5] M. Dorigo, L. Gambardella, "Ant colonies for the Traveling salesman problem." Biosystems 43, pp. 73-81, 1997.

[6] Kamil Rocki, Reiji Suda. Accelerating 2-opt and 3-opt local search using GPU in Travelling Salesman Problem. IEEE 2012.

[7] Molly A.I'Neil, Dan Tamir, and Martin Burtscher. A Parallel GPU Version of the Traveling Salesman Problem.

[8] Kamil Rocki, Reiji Suda. An Efficient GPU Implementation of a Multi-Start TSP Solver for Large Problem Instances.

[9] Giovanni Giardini. Tamas Kalmar Nagy. Performance Metrics and Evaluation of a Path Planner based on Genetic Algorithms.

[10] Thang N. Bui and Mufit Colpan. Solving Geometric RSP with Ants.

[11] Julio Ponce[1], Francisco Ornelas[1], Alberto Hernández[2], Humberto Muñoz[1], Alberto Ochoa[3], Alejandro Padilla[1], Alfonso Recio. Implementation of a Parallel Ant Colony Algorithm Using CUDA and GPUs to Solve Routings Problems Problem.