Jared Reed

Tim Menninger

CS179 Final Project Proposal

GPU Accelerated Traveling Salesman Algorithm

Project Summary:

We intend to use GPU parallelism to speed up the runtime of the traveling salesman problem (TSP). This input to this problem is a weighted undirected graph and the desired output is the smallest-weighted graph in which each node has exactly two incident edges. The brute force approach to this algorithm runs in O(n!), where n is the number of vertices, but with dynamic programming and the Held-Karp Algorithm this can be decreased to only O(n22n). Although this is an improvement, this is still a relatively slow algorithm, which is why we believe it is a prime candidate for GPU acceleration.

Explanation/Background:

The most common application of the problem finding a path through various geographical points that requires the smallest amount of traveling. Early formulations of the traveling salesman problem were known as the messenger problem because mathematicians thought the solution would be most relevant to postmen who wanted to deliver mail while walking as little as possible. It soon became more well-known as the “traveling salesman problem” because it more generally applied to a traveling salesman maximizing efficiency, traveling as little as possible while selling as much as possible. However, this problem can apply to almost any real-world problem that consists of traveling through a set of destinations such that you end up where you began.

The particular application that interests us is the case of a road trip. Between graduation and beginning work, we would like to take a road trip. Prior to this, we would select all of the destinations around the U.S. that we would like to visit during our road trip. Obviously, having not started work yet, we need to mitigate costs. The first way we would do this is by buying as little gas as possible, which means driving as little as possible. This is a goal that can be accomplished by finding the shortest path that connects all of these waypoints and finishes us where we started (be it Caltech or otherwise).

The problem has been studied since the 1800s and falls into a class of problems known as NP-complete. This is a set of problems to which no “fast” solution is known, meaning that there are no known solutions that run in polynomial time. In 1962, Held and Karp proposed a solution using dynamic programming that ran in O(n22n) time on number of vertices, which was a vast improvement from the previous O(n!) solution. We plan on implementing this Held-Karp algorithm first on a CPU and then parallelizing it to run on a GPU. Then we will observe the time speedup for different-sized input graphs, optimizing the GPU code to make this time speedup the greatest.

Challenges:

The TSP is a problem that has been solved numerous times before using a CPU implementation, especially the Held-Karp algorithm which changes it from an almost unfeasible problem to one that can be solved within...years. In these instances, multiple processors were used, so there had to be some level of parallelization. However, given that we are not supposed to reference GPU code for our project, our Google searches were perfunctory at best. This restriction on outside code leads to some interesting and challenging problems for our implementation using cuda. Throughout this course we have never used GPU code to solve an algorithm which builds a solution from the bottom up using memoization. None of our solutions from past problem sets have needed the results of a past thread in order to perform their own thread. This makes it seem like a restriction that makes this problem non-parallelizable, as the need for results of past threads makes it seem like a problem that is solved in sequence, but this is not entirely true. When you look more in depth at the algorithm you see that the subproblems are not branching out linearly, but rather into a tree. This means at the furthest down level we will have many different independent problems to solve, and only as we get close to the solution does it because more single threaded. While this algorithm is fairly simple using CPU code, this introduces a slight issue for GPU code as there is the issue over multiple threads running at the same time. If one thread is attempting to solve a sub-problem that is currently being solved by another thread this is a waste of resources. Therefore the issue that arises is ensuring that each thread is working on its own independent sub problem in the tree without overlapping the work of any other thread. If this is done properly then we expect to see a significant increase in performance of the algorithm over the simple CPU implantation.

Goals:

The goals for this project are fairly simple, as we are only attempting to optimize an algorithm using GPU programming. We will already know the optimal solution generated by the algorithm via the CPU implementation which we will have working before we attempt to optimize it in cuda. As a result, the only goal of this project will be to successfully implement the GPU version of this algorithm which significantly decreases the runtime, especially for large inputs. We will time both runtimes and compare the differences for different input sizes. We will strive to make the GPU version as fast as possible to maximize the difference between CPU and GPU runtime. Throughout this course we have learned many different methods of optimization, and we plan to employ as many as we can that will improve algorithm runtime.

Week-by-Week Timeline:

5/20/16 - Fully implemented CPU version of Held-Karp Algorithm, complete with recorded runtime

5/27/16 - Initial GPU version of Held-Karp Algorithm, complete with recorded runtime. At this point in time it will be ok if it is not running as optimal as possible (i.e no shared memory/ no multi-gpu implementation).

6/3/16 - Fully implemented GPU version of Held-Karp Algorithm, complete with documentation, test cases, and recorded runtime/speed-up analysis. Final write up must also be complete.