**A Case Study in the Performance Benefits of Parallel Computing Using The Travelling Salesman Problem**

**Christopher Foytek, Daniel Quintanilla**

Department of Computer Science, Texas State University, San Marcos, TX

**Abstract** - For this report, we will depict differences in comparable performance measures between two traveling salesman problem (TSP) solvers. The TSP solver we developed is a 2-opt sequential solver, and the other TSP solver is a parallelized multi-start IHC solver developed by O’Neil et al.1 As expected, the parallelized TSP solver vastly outperforms our sequential solver in runtime and speedup, but accuracy of found solution tends to be lower for the parallel solver. We propose two possible mechanisms for the lower accuracy that can be explored in further research.

**Keywords**: TSP, 2-opt, random restart solver, runtime, speedup, accuracy, CPU cycles, Core utilization

**1. Introduction**

The symmetric traveling salesman problem (TSP) has long been one of the most studied problems in optimization. The question asked in the TSP is best stated as “Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?” The problem is often thought of as a complete, undirected, weighted graph, with solutions to the problem given as a Hamiltonian cycle that starts at a vertex *v*, and traverses through all other vertices in the graph, finishing at the starting vertex *v*. An optimal solution to this problem is given by a Hamiltonian cycle through which the minimum distance is traveled.

According to wikipedia, the TSP is labeled as NP-hard; which means that it is a decision problem that is at least as hard as the hardest problems in NP (nondeterministic polynomial acceptable problems) 3 . This means that as the number of cities in the map being optimized increases, the difficulty and runtime of a program attempting to solve for an exact solution also increase dramatically. To avoid these issues, developers use heuristics to find approximate solutions quickly for maps with a large number of cities. These techniques employ multi-start search algorithms that iteratively construct a starting graph, and then improve the graph using heuristics until it cannot be improved further. We examine one such algorithm, iterative hill climbing, with a 2-opt local search improvement heuristic.

The advancement in graphics processing units (GPU) and associated software over the past decade has given consumers the capability to use their relatively cheap graphics cards to execute massively parallel workloads extremely efficiently. These advancements render completely within the realm of possibility problems that were intractable except on the most powerful supercomputers in the year 2000.

Advancement in GPUs alone is not enough to reap performance benefits from parallel computing. The problem that one is attempting to solve must be parallelized, or broken down into many discrete problems with minimal interplay between one another. These discrete problems can then be independently executed by multiple threads on multiple streaming multiprocessors in the GPU, converging at the end of execution with a final solution.

The multi-start IHC algorithm with 2-opt local search is a promising candidate for parallelization because each IHC step begins with a randomized starting graph that is independent from other graphs being tested. It can be parallelized further by parallelizing the 2-opt improvement heuristic, since each 2-opt improvement in and IHC step is also independent from one another.

In this paper we will compare the performance benefits of a fully parallelized multi-start IHC TSP solver with 2-opt search with a fully sequential TSP solver that uses essentially the same algorithm. We produce a sequential solver that iteratively randomizes and improves a tour for a given amount of iterations, and compare it against a parallelized TSP solver produced by O’Neil et al that employs high parallelization to more quickly solve the same TSP with the same number of iterations.

**2. Related Work**

O’Neil et al1. produced a paper titled *A Parallel GPU Version of the Traveling Salesman Problem* in which they parallelize a random-restart iterative hill climbing algorithm with 2-opt improvement heuristics creating a “worklist” of IHC steps to perform, and then assigning “climbers” to threads on the GPU to evaluate possible 2-opt moves in parallel. However, parallel evaluation of 2-opt moves requires thread synchronization at the end of the evaluation to determine the move that will give the best improvement before the tour can be evaluated again for the next-best 2-opt move. The solver also stores the distances between cities in a distance matrix, requiring O() storage. The small shared memory of the GPU limits the input size of this implementation to only 110 cities.

The storage limitation of O’Neil et al.’s TSP solver makes it unsuitable for a comparison to a single threaded solver. Even though there is a drastic increase in performance when compared to a single threaded solver, the single threaded solver can still solve a TSP with 110 cities in a reasonable amount of time.

O’Neil et al2 . produced a second paper in which they address input size, and include many more optimizations to further parallelize and increase performance of the solver. First, instead of pre-calculating a distance matrix, and performing 2-opt evaluations on it, the distances between city coordinates are recalculated each time they’re needed. This avoids a O() space requirement, but also requires more operations for each 2-opt evaluation, since each distance must be calculated when it’s needed. This new solver also exploits the hierarchical nature of parallel processing in GPUs by assigning a climber to each block of threads rather than each thread being assigned a climber. Each thread within a block then evaluates the possible 2-opt moves in parallel and determines the best one. Once the best 2-opt move is found, one thread in the block performs the 2-opt move on coordinate arrays held in local memory, and then the process is repeated until the climber (block) finds a local minimum. This optimization provides for more parallelization and increases overall performance.

These optimizations in the second TSP solver produced by O’Neil et al. provide a promising candidate for performance comparison to a single threaded solver since speedup can be evaluated as input size increases, and since the parallel solver being compared is highly optimized for the algorithm being used.

**3. Implementation**

Our code implements random-restart IHC with 2-opt heuristics by iteratively performing a given number of IHC steps. Each IHC step copies the two global coordinate arrays into local arrays, then randomizes the order of the cities to create an initial random tour. We then loop through each city i, and calculate the change in distance if a 2-opt move were performed on i with all other cities j in the tour to find the best initial 2-opt move to perform. When a best move is found, it is performed, and then the next best 2-opt move is found iteratively until a local optimum is found for the tour. If the found optimum is better than the last best optimum, it is stored, and the next IHC step is performed until all steps have been performed.

The bulk of the operations performed in our algorithm occurs in finding the local optimum for each tour. The number of improvements each tour will need to reach a local optimum from its starting state varies depending on the starting order, but a lower bound for the complexity of this algorithm is at least O() assuming that no improvements are needed to find a local optimum and only one IHC step is performed. However, despite the computational difficulty of this implementation, it is a good comparison to the parallel TSP solver it is tested against, since it employs the same algorithm performed sequentially.

**4. Evaluation Methodology**

To perform our comparison tests, we use a computer with a 4-Core Intel Core i5-6600K CPU clocked at 3.5 GHz, a Samsung 850 EVO SSD, 16GB of DDR3 RAM, and an Nvidia GeForce GTX 960 GPU clocked at 1127 MHz.

Since our sequential TSP solver is written in C, we compiled with gcc and utilized the math library, -lm, and since the parallelized TSP solver is written in CUDA, we compiled with nvcc. In order to establish a solid dataset for our evaluations we used 7 different files, and the number of coordinates per file range from 150 – 2392. We form our result set by running each input file in both the sequential and parallel TSP solvers with 25 IHC steps. We record data for runtime and minimum tour cost found by each solver, and use runtime to calculate the speedup granted by the parallel solver over the sequential solver. Runtime recorded is only the runtime of the IHC steps performed in each solver. This allows us to avoid overhead from file read, initial memory allocation, and output that is trivial to performance of the algorithm.

By running each solver with 25 IHC steps, we can produce a fair comparison of the accuracy of each solver, while still maintaining a feasibly short runtime. However, due to the nature of greedy randomized adaptive search algorithms, the accuracy is dependent on the number of steps performed. We do not expect either solver to be wholly accurate or find optimal solutions for the given input, but we can compare the relative accuracy of each solver given the same small number of IHC steps.

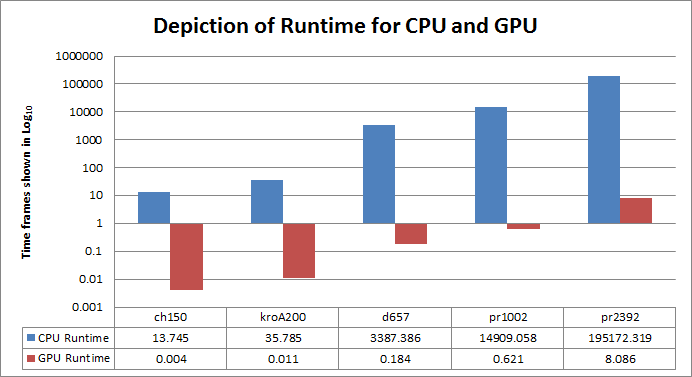
**5. Results**

Figure 1 shows raw data collected from both IHC implementations. The input input files are TSPLIB inputs, and the number attached at the end are the number of city-coordinate pairs the file contains. MinCost is the minimum tour cost found by each solver after 25 IHC steps, and Runtime is recorded in seconds. The results show a dramatic difference in runtime between the GPU and CPU solver even with small file sizes. All inputs tested with the CPU implementation required at least several seconds for a solution to be found, while the GPU solver found a solution in less than a second for four out of five inputs. We also see that the GPU and CPU solver both seem to find similar values for the minimum cost of each tour.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **CPU** | | **GPU** | |
| **Input File** | **MinCost** | **Runtime** | **MinCost** | **Runtime** |
| ch150 | 6918 | 13.745 | 6846 | 0.004 |
| kroA200 | 30057 | 35.785 | 30922 | 0.011 |
| d657 | 50397 | 3387.386 | 52603 | 0.184 |
| pr1002 | 277531 | 14909.058 | 281078 | 0.621 |
| pr2392 | 394361 | 195172.319 | 419760 | 8.086 |

**Figure 1. Minimum cost and runtime (in seconds) of both CPU and GPU implementations**

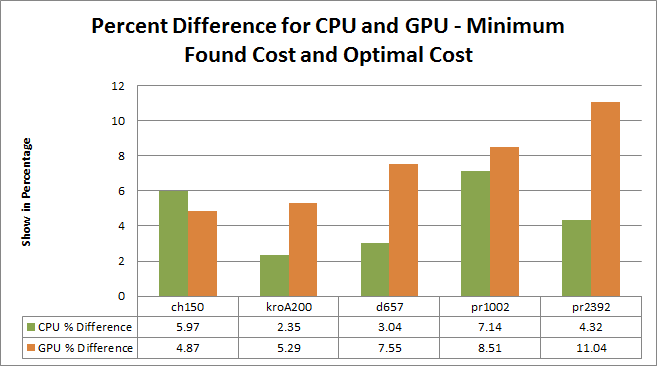
Figure 2 displays the amount of time in seconds that each implementation took to find a solution for each file. The graph is logarithmic, so the actual differences in runtime are much greater than shown. Regardless, the graph allows us to visualize the very large difference in runtime between the implementations, especially as input size increases. Note that both the CPU and GPU implementation experience a sharp increase in runtime between files kroA200 and d567, and again between files pr1002 and pr2392. This is likely due to the large increase in size between the inputs. If a smoother increase in input size were used, we would likely see a smoother increase in runtime. However, at the time of writing, no appropriate inputs for these solvers that increase more linearly exist on TSPLIB.



**Figure 2. The runtime (in seconds) for each input. Note that this graph is logarithmic and the actual differences between runtimes is much larger**

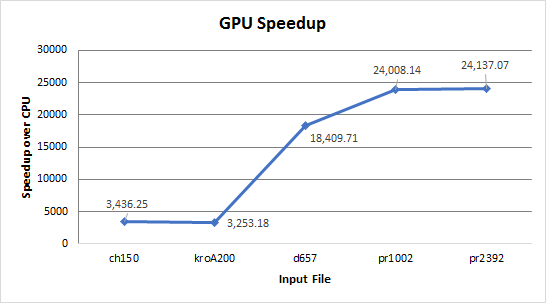
Figure 3 shows the accuracy of each solver by displaying the percentage of difference between the minimum tour cost found for each input and the known optimal tour cost of each input. The results show that the CPU implementation outperforms the GPU implementation for all but the smallest input. This result is somewhat surprising given that both solvers use the same algorithm to find a solution. A proposed cause of this discrepancy may lie in how tours are initially randomized in each IHC step. The GPU implementation randomizes the tour using the global thread index as a seed for generating random array indexes, whereas the CPU implementation uses the current UNIX time as a seed. The UNIX time is a much larger seed and as such, may produce a more random sequence of indexes each time the tour is randomized. Because the tour is more random, it is more likely to to be further away from other, possibly already discovered local minimums in the solution space, and therefore each IHC step is more likely to greatly improve the minimum tour cost. However, more testing is needed to determine if this is the cause of the higher average accuracy of the CPU implementation.

The higher accuracy of the CPU implementation may also be explained by the nature of greedy randomized adaptive search algorithms. Since the starting state of each tour is inherently random, the appearance of higher accuracy in the CPU implementation may be only a matter of chance that it found starting tours that were closer to the global maximum. A large sample size created by repetitive testing with each implementation is needed to determine if this difference can be averaged out, or if there is a more complex mechanism causing the difference.



**Figure 3. The percent difference between the found minimum tour cost for both implementations and the optimal tour cost for the input**

Figure 4 shows the speedup achieved by the GPU implementation relative to the CPU implementation. We again see that the GPU implementation far outperforms the CPU implementation even for small input sizes. However, note that the speedup also increases as the file size increases, eventually leveling out to about a 24,000X speedup with inputs larger than 1000 cities. This increase in speedup as file size increases is logical as the GPU implementation is more effectively able to harness parallelization as input size increases since each IHC step will have more 2-opt moves that can be performed in parallel. This is true up to a certain point at which the GPU implementation can no longer allocate more threads to find 2-opt moves in parallel, and must start waiting for a thread to finish before it can be assigned to test another 2-opt move. However, this effect still produces a very large speedup since the CPU implementation must perform all 2-opt move tests sequentially.



**Figure 4. The speedup of the GPU implementation relative to the CPU implementation for each input file**

**6. Conclusion**

This report illustrates the differences in comparable performance measurements we found between our two specified TSP solvers. Our results depict an extreme amount of speedup with our GPU parallelized solver because of the incredibly short amount of runtime compared to the sequential CPU solver. However, we can also conclude from our report that the CPU solver is more accurate than our GPU solver based on found minimum tour cost for each solver. We believe that, unless time is of no importance when utilizing our two IHC TSP solvers, despite the slightly lower accuracy the parallelized TSP solver will be the best option based on the astounding amount of speedup provided. This lower accuracy can also be offset by running the GPU solver with more IHC steps, as the performance of the GPU implementation is not affected near as much by increasing the number of IHC steps.

This paper falls slightly short in its analysis by comparing an unoptimized sequential solver to a fully optimized parallel solver. There are several optimizations that can substantially increase performance for the sequential solver. For example, the GPU implementation uses coordinate arrays to store the city coordinates, and then re-calculates distances between cities when needed. This is a necessary choice for the GPU implementation because if the cities were stored in a distance matrix, it would limit the size of the input due to the small size of shared memory on the GPU. However, the performance impact is minimal since GPU architecture is optimized for fast calculation, and therefore distances can be re-calculated as needed. In the CPU implementation, we also re-calculate distances in real time, but the performance impact is much greater as the CPU is not as optimized for mathematical operations. It is also unnecessary to re-calculate distances on the CPU, most computers have more than 4GB of RAM, and therefore an O() storage requirement for a pre-calculated distance matrix is a non-issue. This optimization would greatly reduce time spent performing both memory access and floating point calculations, both of which are very expensive operations on the CPU.

Another small optimization that could improve performance is to copy the coordinate arrays or distance matrix once before performing any IHC steps, and then randomize from the state of the matrix or arrays at the end of each IHC step. This saves unnecessary memory reads and writes at the beginning of each IHC step. However, it is unknown what effect this change would have on accuracy.

With these and other optimizations, future work may find a significant increase in performance for a sequential TSP solver, but we predict that even a fully optimized sequential solver cannot match the performance of a parallelized solver using this algorithm.

**7. References**

[1] O’Neil, M. A., Tamir, D., & Burtscher, M. (2012). *A Parallel GPU Version of the Traveling Salesman Problem.* Retrieved February 20, 2017, from  
<http://weblidi.info.unlp.edu.ar/worldcomp2011-mirror/PDP3801.pdf>

[2] O’Neil, M. A. & Burtscher, M. (n.d.). *Rethinking the Parallelization of Random-Restart  
Hill Climbing A Case Study in Optimizing a 2-Opt TSP Solver for GPU Execution.* Retrieved February 20, 2017, from <http://cs.txstate.edu/~mb92/papers/gpgpu15a.pdf>

[3] Traveling Salesman Problem. (n.d.). In Wikipedia. Retrieved February 20, 2017 from<https://en.wikipedia.org/wiki/Travelling_salesman_problem>.

[4] G.C., Mauricio, Resende, and Celso C. Ribeiro. "Greedy Randomized Adaptive Search Procedures." *Handbook of Metaheuristics* (2003): 219-49. Print.