GPU Acceleration of 2-Opt for the Traveling Salesman Problem

Robert Lee

Project Report

CSE6230

Fall 2014

Background

The traveling salesman problem is one of the most studied well known NP-Hard problems. Given a set of cities, the problem is finding the shortest route that visits all cities and returns to the origin. One possible way to solve this would be to enumerate all possible routes and choose the best. This is however of factorial complexity. A problem with 60 cities would have more routes than there are atoms in the observable universe. Obviously, brute force is not very feasible.

Fortunately, many successful approximation methods have been developed for the traveling salesman problem. It is common to solve within a few percent of optimality in reasonable time using reasonable resources. A very well-known method is called 2-opt, where two edges are swapped with another set of lower-cost edges that do not exist in the tour, but upon swap will still result in a valid tour of lower cost. The 2-opt method is very simple, yet effective. There are many variants that involve tradeoffs between accuracy and run time. The k-opt algorithms are also used as auxiliary methods in other, more sophisticated algorithms such as Genetic Algorithms and Simulated Annealing [2].

A generalization of the 2-opt is the k-opt, where k is a number greater than or equal to 2. 3-opt is the more accurate but more computationally expensive sibling of 2-opt. The higher the value of k, the more expensive but more accurate the method becomes.

One of, if not the, most successful approximation methods in existence is the Lin-Kernighan algorithm and its variants. It employs a ‘variable’ k-opt, where k changes according to some criteria as it performs its local search. It is much more complicated than the ‘static’ k-opt algorithms mentioned before, but also more effective. The Lin-Kernighan-Heldsgaun 2 variants contribute to the frontier of (increasingly large) unsolved traveling salesman problems.

Although there are more sophisticated and effective algorithms out there for some problems, the 2-opt is still a valuable method due to its simplicity, effectiveness, and applicability within other algorithms.

Implementing 2-opt on a GPU has been done before with good results by Burtscher [1] and Rocki [3]. It will be shown that potential improvements are still available to be made to build upon this prior work. The code can be found at the following repository:
https://bitbucket.org/lordvon/finalproject

Algorithm

The 2-opt algorithm searches all possible edge swaps to find the best improvement (decreases cost the most) or first improvement (first found to decrease cost). The swap is then performed and the procedure is repeated on the new, post-swap tour. This procedure can be repeated until no more proving swaps can be performed. The tour is then at a local optimum. To try to find better local optima or the global optimum, the tour can be perturbed in a non-greedy way, ranging from random restarting of the whole tour to a series of random swaps.

The 2-opt search-and-swap process for the symmetric traveling salesman problem is commonly done by a simple double for loop. There are several ways to handle indexing and swaps, and the listing below shows one possible way, for the first-improvement heuristic. A best-improvement implementation looks similar, but runs for the entire for loop and records the indices and score of the best improvement, performing the swap after the double loop has been exhausted. Note that it is less than NC^2, where NC is the number of cities, because of the symmetry of distances and edges.

```
1  //NC is the number of cities, with ids from 0 to NC-1.
2  for(int i=1;i<NC-2;++i)
3  {
4      for(int j=i+1;j<NC-1;++j)
5      {
6          EDGE oldEdge1 = edge(i-1,i);
7          EDGE oldEdge2 = edge(j,j+1);
8          EDGE newEdge1 = edge(i-1,j+1);
9          EDGE newEdge2 = edge(j,i);
10         if(cost(oldEdge1)+cost(oldEdge2) > cost(newEdge1)+cost(newEdge2))
11         {
12             swap(i,j);
13         }
14     }
15 }
```

Figure 1: A listing of a typical for loop for first-improvement checking of 2-opt swap possibilities.

The figure below shows a mapping of the unique swaps that can be checked.

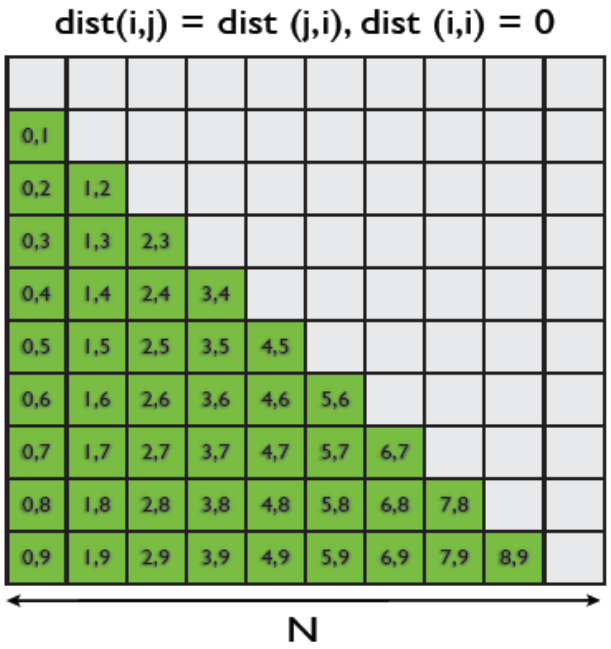

Figure 2: Mapping of possible swaps. The entire square represents all permutations, and the green regions represent the unique pairings in the context of symmetric TSP.

Implementation

In CPU implementations, the costs of all possible edges are typically stored in a distance matrix. The number of edges possible is O(NC^2), where NC is the number of cities. Storing edge costs saves computation by avoiding repeated computation of edge distances.

However, once this strategy is in place, the computational density is very low. GPUs thrive on computationally dense tasks. The distance matrix increases the memory requirements, while the decreasing the FLOP count. Rocki [3] does not construct a distance matrix, only transferring coordinates (O(NC)), and has found that repeated computation of distances in return for smaller memory transfers actually increases speedup!

Another issue for GPU implementation is the work division. The listing shown earlier is well and good as a sequential process, but obtaining an index mapping for parallel work with minimal idle threads requires a transformation. The following figure shows an indexing transformation that allows one to derive the proper 2-D coordinates from serialized / 1-D indices. This is non-trivial because of the triangular shape of the space of unique swap possibilities. Using only 2-D indices would result in roughly half of the indices being redundant / idle.

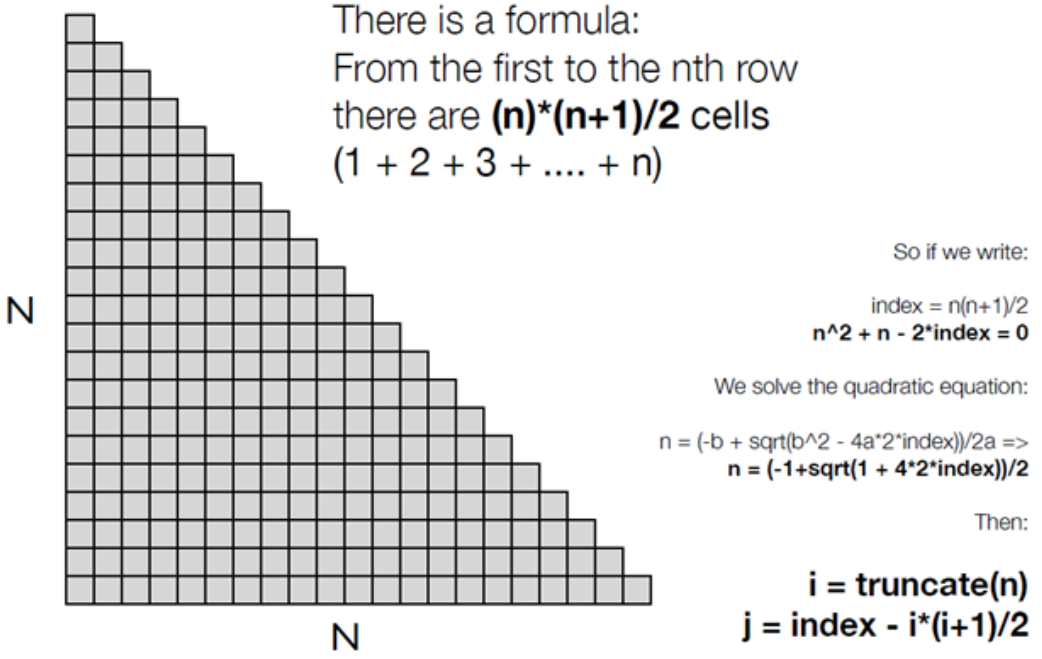

Figure 3: Indexing 1D to 2D indexing of a triangular map. From a presentation by Rocki.

The two main references in GPU implementation of 2-opt is Rocki [3] and Burtscher [1]. The first implementation of Rocki assigns a unique tour to each thread, where each thread performs a 2-opt search-and-swap as a sequential processor would. This approach does not utilize shared memory and all memory transactions are essentially to and from the global memory. Two straightforward optimizations were implemented: transferring the city coordinates to shared memory, and ordering the coordinates

(in order of the tour) before transfer to the GPU. The latter allows for coalesced global memory transactions, which offer large returns on speed for little overhead cost in ordering the coordinates. The shared memory option was explored in two ways: transferring the whole coordinate set to shared memory, and breaking up the coordinates to be processed piecewise. Obviously, transferring the whole coordinate set is very limiting for the maximum number of cities (limited by the shared memory size). Breaking up the computation through tiling is a more scalable method, and, as shown by Burtscher, faster due to no shared memory bank conflicts.

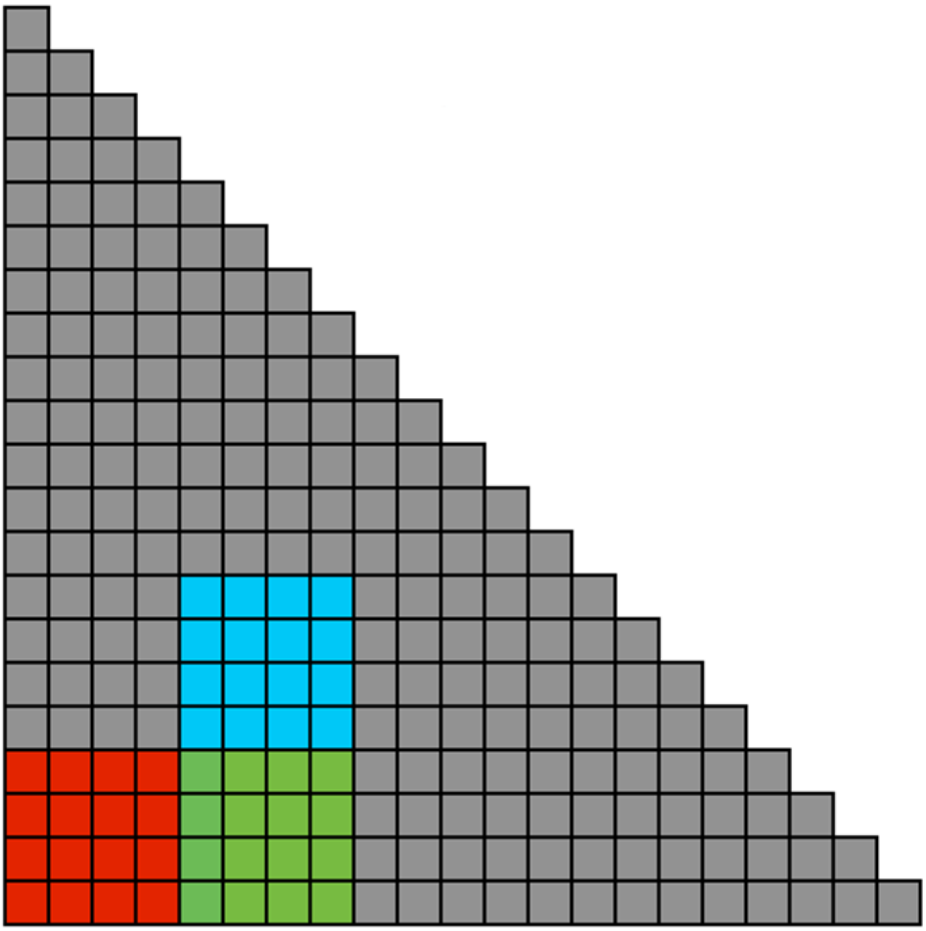

Figure 4: Visualization of tiling of the swap map from Rocki [3].

I attempt to further improve the performance by increasing the placing more memory directly into registers, and increasing the number of swap checks performed per thread. This requires tiling not only at the block level, but also at the thread level.

I develop several versions of the 2-opt kernel, using a min reduction that utilizes shared memory to find the best swap. The reduction step remains the same through the different versions, so differences in speed can be attributed to the algorithm changes. The following table summarizes the kernels and their improvements.

Table 1: Kernel versions implemented.

Kernel Version	Description
1	Naïve, straightforward implementation using global memory. One swap check per thread.
2	Version 1, but with a minimum reduction step.
3	Version 2, but utilizing shared memory tiling.
31	Version 3, but rearranging shared memory allocation and access to help with bank conflicts.
4	Version 3, but instead of shared memory tilling, register / local memory tiling is used.
5	Version 4, but skipping the GPU minimum reduction.
6	Version 4, but with loops unrolled so that registers are actually used, instead of local memory.
7	Version 6, but pre-computing distances within the two city sets.
8	Version 7, but pre-computing even more distances, within and between the two city sets.

Martin Burtscher kindly provided his code for comparison purposes. However, the code structure is significantly different (including his method of reduction to find the best swap, use of atomic functions, and use of Kepler architecture), so a direct comparison may not be valid. The structure of his implementation allows shared memory to be used more effectively than in kernel 3, via multiple iterations in one kernel call.

Results and Discussion

The following results are for a single 2-opt best improvement search, on u2319.tsp from TSPLIB. This means 2,683,086 swap checks were performed per iteration. The Intel Xeon X5650 processor and Tesla M2090 GPU on the Jinx cluster were used. The iteration without the CPU reduction includes the swap check calculations and GPU minimum reduction. The iteration with CPU reduction includes swap check calculations, GPU minimum reduction, GPU to host transfer, and CPU minimum reduction of the transferred dataset.

The sequential version uses a distance table, which on the CPU is more efficient than re-computing distances.

Table 2: Kernel runtimes for a single 2-Opt iteration of 2,683,086 swaps.

Kernel	Runtime without CPU Reduction (milliseconds)	Speedup over Sequential (GPU without CPU reduction)	Runtime with CPU Reduction (milliseconds)	Speedup over Sequential (GPU + CPU Reduction)	Speedup over Kernel 2
Sequential (with distance table)	130	1	128	1	0.004727
1	0.435821	298.3	13.056	9.8	0.04634
2	0.537901	241.7	0.60501	211.6	1
3	0.690963	188.1	0.75750	169.0	0.7987
31	0.689536	188.5	0.75727	169.0	0.7989
4	0.293946	442.3	0.31679	404.0	1.910
5	0.287866	451.6	0.86298	148.3	0.7011
6	0.294714	441.1	0.31782	402.7	1.904
7	0.200531	648.3	0.22388	571.7	2.702
8	0.169901	765.2	0.19248	665.0	3.143

The naïve implementation using only global memory (kernel 2) with only one thread per swap and no GPU reduction still gives an order of magnitude speedup. Storing in shared memory did not give performance benefits vs pulling directly from global memory, but it should be noted that the shared

memory implementations can perform better if the kernel better utilized the stored shared memory by performing multiple iterations on the same shared memory (a much more complicated implementation).

The register tiling resulted in significant improvement in the runtime over shared memory or loading directly from global memory. Reusing data stored in registers reduces the global memory transactions.

Kernel 6 was expected to give an even greater improvement, but it did not occur. Compiler messages showed that in kernel 4, data intended for registers were being stored in local memory, due to the compiler being unable to store them in registers because non-constant numbers were used in the indexing of these register (in code) arrays. This problem was solved by unrolling all of the loops that used the register arrays. Unfortunately, no improvement was seen in kernel 6. This means that the memory access speed for the data stored in the registers was not a bottleneck. This realization motivated the development of the next kernels, which reduce the amount of re-computation.

Kernels 7 and 8 embody the key improvements over previous implementations of 2-Opt GPU in literature. Assigning a tile to a thread facilitates the opportunity to avoid some re-computation by constructing a small distance table relevant to a certain thread and storing this table in registers.

References

[1] M. A. O'Neil, D. Tamir, and M. Burtscher. A Parallel GPU Version of the Traveling Salesman Problem. 2011 International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 348-353. July 2011.

[2] Fujimoto N, Tsutsui S, Dimov I, Dimova S, Kolkovska N. A Highly-parallel TSP Solver for a GPU Computing Platform. [serial online]. January 1, 2011;Available from: Inspec, Ipswich, MA. Accessed December 5, 2014.

[3] Rocki K, Suda R, Balaji P, Buyya R, Majumdar S, Pandey S. Accelerating 2-opt and 3-opt local search using GPU in the travelling salesman problem. [serial online]. January 1, 2012;Available from: Inspec, Ipswich, MA. Accessed December 5, 2014.