Analysis of a High-Performance TSP Solver on the GPU

JEFFREY A. ROBINSON, SUSAN V. VRBSKY, and XIAOYAN HONG, University of Alabama
BRIAN P. EDDY, University of West Florida

Graphical Processing Units have been applied to solve NP-hard problems with no known polynomial time solutions. An example of such a problem is the Traveling Salesman Problem (TSP). The TSP is one of the most commonly studied combinatorial optimization problems and has multiple applications in the areas of engineering, transportation, and logistics. This article presents an improved algorithm for approximating the TSP on fully connected, symmetric graphs by utilizing the GPU. Our approach improves an existing 2-opt hill-climbing algorithm with random restarts by considering multiple updates to the current path found in parallel, and it allows k number of updates per iteration, called *k-swap*. With our *k-swap* modification, we show a speed-up over the existing algorithm of 4.5× to 22.9× on data sets ranging from 1,400 to 33,810 nodes, respectively.

CCS Concepts: • **Mathematics of computing → Combinatorial optimization**; *Approximation algorithms*; • **Theory of computation → Vector / streaming algorithms**;

Additional Key Words and Phrases: CUDA, TSP, 2-opt, k-swap, GPGPU

ACM Reference format:
Jeffrey A. Robinson, Susan V. Vrbsky, Xiaoyan Hong, and Brian P. Eddy. 2018. Analysis of a High-Performance TSP Solver on the GPU. *ACM J. Exp. Algor.* 23, 1, Article 1 (January 2018), 22 pages.
https://doi.org/10.1145/3154835

1 INTRODUCTION

The Traveling Salesman Problem (TSP) is a well-studied problem in combinatorial optimization with numerous practical applications found in, but not limited to, the fields of engineering, electronics, and transportation [1, 7, 10]. The TSP can be stated as this: given a starting city, visit all other cities exactly once and then return back to the starting city, visiting it exactly twice, with there being no other such tour that is shorter. Finding such a tour is NP-hard [4] and no polynomial time algorithm is known that can find an exact solution. Approximation algorithms are sometimes used to find near optimal solutions to problems such as the TSP, which sometimes incorporate parallelization such as that provided by GPUs.

The TSP problem can be represented as a graph where the vertices, V, are the cities, and the edges are E, such that an edge from vertex u to vertex v (u, v) represents a road between u and v. Each edge is given a non-negative weight $w(u, v)$. The cost of a tour is given as the sum of the cost

Authors' addresses: J. A. Robinson, S. V. Vrbsky, and X. Hong, Department of Computer Science, The University of Alabama, Box 870290, Tuscaloosa, AL 35487-0290; emails: jarobinson3@crimson.ua.edu, {vrbsky, hxy}@cs.ua.edu; B. P. Eddy, Building 4 Room 223, 11000 University Parkway, Pensacola, FL 32514; email: beddy@uwf.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 ACM 1084-6654/2018/01-ART1 $15.00
https://doi.org/10.1145/3154835

of each edge that is on the tour. In the fully connected symmetric TSP problem $w(u,v) = w(v,u)$ and for all vertices $u, v \in V$, there is an edge $(u,v) \in E$.

Some examples of approximate algorithms for TSP are genetic algorithms, ant-colony optimization, self-organizing maps, and the k-opt family of heuristics [1, 5, 11, 20, 22]. Limitations of previous work include larger space requirements, small graphs, either no analysis of different configurations or limited analysis based on a few iterations of algorithms. For instance, the ant-colony optimization approach requires each edge to be pre-computed and saved. However, for 2-opt when using graphs with Euclidian distances, the edge costs can be recomputed and no extra information is required per edge, giving a memory footprint of $O(V)$ instead of $O(V + E)$. 2-opt is a part of the k-opt family and replaces two edges of a graph if it makes the distance shorter. In Reference [20] the authors limited their analysis to the 2-opt search and to a few iterations of each.

Iterative hill climbing (IHC) is another heuristic approach and is commonly applied to the graph TSP problem. This technique works by generating an initial tour, which might not be optimal, and repeatably applying a heuristic until some acceptance criteria has been reached, i.e., reached some iteration limit or no possible update is possible. 2-opt is a common heuristic used in IHC and works by taking two edges, (u,v) and (s,t), in the tour and replaces them with the edges (u,s) and (v,t) if this change makes the tour shorter. In this case, the IHC algorithm compares each pair of edges in the current tour, finds the two edges that result in the best swap, swaps them with the alternative edges, and repeats until there are no new edges that can improve the tour.

The 2-opt algorithm does not guarantee the optimal solution and can converge to local optimal solutions. However, the IHC 2-opt algorithm tries to mitigate this by randomly generating tours and applying the 2-opt algorithm to it, allowing for the 2-opt algorithm to potentially find better tours. The worst case running time of the 2-opt algorithm in the Euclidean plane is exponential, but the average runtime is shown as $O(n^6)$ [6].

One approach to maximize the efficiency of approximation algorithms is to take advantage of the parallelism provided by GPUs. Originally, GPUs were designed and developed as highly parallel architectures for rendering graphics with a fixed pipeline for execution. The inclusion of a programmable pipeline for GPUs allowed for researchers to develop more general purpose programs using a domain-specific shader language, the first such program being matrix multiplication [9]. Initially, general purpose graphics processing unit (GPGPU) applications were difficult for which to develop programs, so researchers started writing new APIs and languages to support general purpose programming [3, 12]. With the introduction of the OpenCL and CUDA APIs, the landscape changed tremendously [13, 18]. These two APIs can be used in multiple programming languages while providing access to the underlying GPU hardware. As a result of the introduction of GPGPUs, there has been a rush to move parallelizable algorithms to the GPU, e.g., machine learning, optimization, databases [2, 14, 19], and graph problems.

Our work takes a previous implementation of an IHC 2-opt algorithm developed for CUDA GPUs, which was an extension of the authors' previous work [15], and modifies it to improve performance [14]. We decided to extend this work due to its high parallelism and computational work along with having the theoretical lower bound on the space complexity of $\Omega(V)$. The motivations for this article are as follows: the extensive number of problems related to the TSP, the GPU provided a high-performance opportunity, and previous GPU results imposed unneeded limits and used small input sizes. Problems, such as circuit paths, are currently getting larger and require more efficient algorithms. This leads to parallel solutions due to the limitations of modern hardware. We chose the GPU due to its costs for the performance provided, natural fit between the algorithm and architecture, and relatively limited amount of research on the GPU and graph TSP problem. We decided on this implementation to analyze because at the time of this writing it

appeared to be one of the fastest TSP approximation algorithms for the GPU. We also found limitations imposed by the algorithm that can lead to how close we can get to the optimal solution.

We modified the existing IHC algorithm to include the variable k, which represents the number of updates per iteration allowed and call this modification *k-swap*. We also include the impact of the configuration on the modified 2-opt algorithm, specifically the GPU configuration, by considering the number of blocks utilized and introducing a work queue to ensure each block remains busy. We study the effect of the configuration and modifications on the performance of the *k-swap* algorithm. We note that in this article we are not proposing an implementation of the algorithm to provide an optimal solution which requires costly computational resources. Instead, we improve the performance of a heuristic that can provide a faster and more accurate approximation to the optimal solution in a shorter amount of time and at much less computational cost.

In Section 2, we go deeper into the 2-opt algorithm, and in Section 3, we discuss additional modifications to the algorithms and code. The experimental setup along with a description of each experiment and its configuration are discussed in Section 4. Next, in Section 5, we show and explain the results as well as possible areas for improvement. In Sections 6 and 7, we conclude and describe future work.

2 BACKGROUND

This section provides necessary background information for the Traveling Salesman Problem (TSP), 2-opt algorithm, and CUDA architecture. We also discuss some of the limitations of the 2-opt GPU algorithm.

2.1 The Traveling Salesman Problem

The Traveling Salesman Problem can be described as such: given a set of cities connected by roads, in what order should a salesman visit each city such that he leaves his home, visits each city exactly once, and finally returns home. An additional constraint is that in such a path the distance traveled or cost is minimized. More formally, Let G be a connected symmetric graph with a set of vertices V, edges E, and weight function $f : E \rightarrow \mathbb{R}$, where f$(v, v) = 0$. To form a tour T on G, we visit each vertex in V such that the first and last vertex visited is the same and all other vertices have only been visited once. The tour T can be formulated as an ordered set of vertices such that if two elements are adjacent, v_i and v_{i+1}, then the second was visited directly following the first and the edge (v_i,v_{i+1}) was used to get there. The length of T is denoted as the sum $\sum_{v_i \in T} f(v_i, v_{(i+1 \mod (|V+1|)})$ for $i \in \{1, 2, \ldots, |V|\}$. Solving the Traveling Salesman Problem is finding a Hamiltonian circuit of a complete graph with minimal cost of the weighted edges.

2.2 2-Opt

The 2-opt heuristic of the Traveling Salesmen Problem works by taking two edges from a given tour and replacing them with two different edges such that it reduces the length of the tour [5]. In the algorithm, all possible pairs of vertices are examined and exchanges or swaps between pairs of edges are considered. For each pair of edges, if swapping the two edges provides a shorter tour, existing paths are swapped and the edges reordered. The algorithm then considers the next pair of edges. The algorithm stops when no new pairs that can decrease the tour length are found.

The 2-opt TSP utilizes a random iterative hill-climbing approach that starts from a randomly generated candidate tour. The 2-opt TSP algorithm iterates until no more swaps can improve the tour, in essence until a local optimum has been found. A hill-climbing strategy will become stuck in a locally optimal solution, requiring a new randomly generated candidate tour to begin a new local search. Once the current best tour has been found and recorded, the strategy *restarts* from

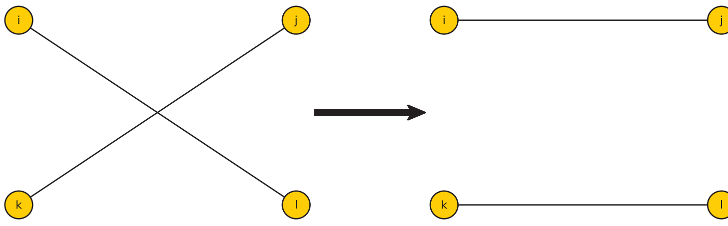
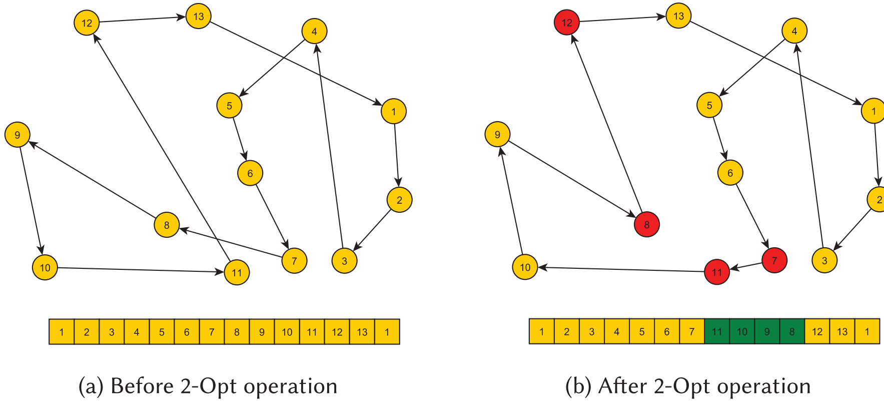


Fig. 1. A 2-opt swap.



(a) Before 2-Opt operation

(b) After 2-Opt operation

Fig. 2. Tours before and after a 2-opt operation. Red denotes nodes that were reconnected; green represents elements that were reversed.

a new randomly generated candidate tour so a new climb can begin. After each climb, the local optimum obtained is compared to the shortest tour found thus far, to find a new global minimum.

In Figure 1, the edges (i, l) and (j, k) can be swapped with the edges (i, j) and (k, l). This swap results in paths that no longer cross. The 2-opt algorithm utilizes the triangle inequality to reduce the length of the tour, i.e., reordering edges for paths that cross over so that the paths no longer cross. As can be seen in Figure 1, when swapping two edges on a tour there is only one alternative way to reconnect that produces a shorter valid tour.

In Figure 2(a) a more complex graph with 13 nodes is illustrated. The 2-opt algorithm has determined that swapping the edges (7,8) and (11,12) with (7,11) and (8,12) can reduce the tour length. Figure 2(b) shows the resulting changes in the graph and tour set with the changes highlighted. Note that the direction of the edges can also change to preserve a legal tour, resulting in a reversal of the path. When (7,8) and (11,12) are swapped, the path (8,9,10,11) is reversed. After the 2-opt algorithm is applied, the nodes in the path 7, 8, 9, 10, 11, 12 are reordered to create the new path 7, 11, 10, 9, 8, 12.

One of the first scalable parallel 2-opt algorithms was presented in Verhoeven et al. [21], in which a tour is divided among multiple transputers[1] by edges. Edges are swapped by rotating the tour among transputers. On each transputer, it tries to perform the 2-opt swap for each possible edge and then gives an edge to an adjacent transputer while accepting a new edge from another adjacent transputer.

[1]An early parallel system on a chip architecture that was defeated by costs and newer superscalar architectures.

When using GPGPU for parallel algorithms, there are two designs of the 2-opt algorithm. In the first design, the entire algorithm is created on the GPU. In the second design the algorithm is divided into multiple parts where some work is computed on the GPU then updates are performed on the CPU. In Fosin et al. and Rocki et al. [8, 17] the CPU generates an initial path, performs the 2-opt heuristic on the GPU, then returns control to the CPU to apply updates and performs a random swap. This process then repeats. In this article, we modify the work of O'Neil et al. [14], which implemented a GPU version of the 2-opt algorithm. In their work, the authors use shared-memory caching to reduce memory bottlenecks and recompute values instead of storing them in memory.

2.3 NVIDIA CUDA Architecture

CUDA enabled NVIDIA GPUs contain a set of processors, global memory, texture memory, and constant memory, including an L2 cache. Each processor contains its own local resources, e.g., registers, shared memory, local memory, and L1 cache, which are not accessible to other processors. The overall program is at the highest level and contains independent blocks, which handle their own set of resources that cannot be shared outside the block. Each block is assigned to a processor where it resides until it has completed. Blocks are then further divided into warps, which are groups of 32 threads all running in a SIMD fashion.

During execution, each processor will select some number of warps to run from the blocks assigned to it. The hardware limitations of the compute capability determine the number of warps that can run. Each warp is either labeled as ready to execute or blocking. All warps will continue to execute until they have to block, either due to memory fetching or other resource limitations, and any ready warp will be scheduled to execute. Since there are more warps than can be executed, the hardware can hide latencies by swapping out blocking warps with ready warps. If there are no ready warps, then execution halts until a warp is ready. Each thread in a warp is composed of its own local resources, which cannot be shared, as well as resources shared among both threads in a block and resources shared among the entire grid.

To develop applications for CUDA, the user must write a kernel global function that is executed on the GPU device and is called from the CPU host. When calling a kernel function from the host, the user must also specify a configuration that controls how many blocks are created and how many threads are in each block. Optionally, the user can also dynamically allocate shared memory.

3 IMPROVEMENTS

Graph algorithms can pose issues during the parallelization portion of the algorithm as it is not always clear how to partition the work and data. The 2-opt algorithm is one of the few graph algorithms that can be parallelized efficiently, because finding swaps are independent. In this article, we take an existing GPU 2-opt implementation [14] and make changes to improve its scalability and performance. Issues we need to address to provide improvements in the original 2-opt implementation are architectural and algorithmic. After applying some initial modifications to the code related to memory and number of restarts, we address architectural limitations of the original 2-opt implementation by limiting the number of blocks and adding a work queue. We then propose a modification to the 2-opt algorithm, called *k-swap*, that limits the number of updates. In the following subsections, we elaborate the improvements in detail starting with a description of the original algorithm before any modifications. We note that the approach presented in this article is not GPU dependent. The GPU thread blocks can be abstracted for processors or virtual processors, whereby each thread in a GPU can map to a lightweight thread with the appropriate synchronization.

```
1:  BEST = 0 {each thread in a block performs the following}
2:  while BEST ≠ -1 do
3:     BEST = -1;
4:     for i = 0 to cities − 1 up by num_threads do
5:        CITY = tour[i + threadID]
6:        NEXT = tour[i + threadID + 1]
7:        LAST = tour[cities]
8:        EDGE = (CITY,NEXT);
9:        for j = cities − 1 to i + 2 down by 1 do
10:          Load tour[j − num_threads...j] to SHARED
11:          for n = 0 to num_threads do
12:             OTHER = (SHARED[n],LAST)
13:             TMP = change(EDGE,OTHER)
14:             if TMP > BEST then
15:                BEST = TMP
16:             end if
17:             LAST = SHARED[n]
18:          end for
19:       end for
20:    end for
21:    BEST = max( BEST ) {max( ) gets the maximum decrease among all threads}
22:    perform_swap( BEST )
23: end while
```

Fig. 3. Pseudocode for original implementation.

3.1 Original CUDA 2-Opt Algorithm

The code for the original 2-opt algorithm appears in Figure 3. The original implementation consists of a single monolithic function to perform the initial random generation, search, and update phases. Prior to executing the code in Figure 3, the program starts by computing the best possible number of threads per block and the maximum amount of shared memory that it can allocate to each block. From examining the thread count selection algorithm [14], we find that it selects the minimum of either 128 threads or the number of cities in the graph. This version of the code is a self-contained kernel function in which a block copies the original tour to a pre-allocated memory location for the block, permutes the tour using the block id as a seed, and then computes the edge weights. Each block handles a single random graph and assigns the graph to its threads.

The code in Figure 3 is a parallel-for loop at the thread-block granularity where each different thread starts a different random tour. The variable BEST is a thread local variable (lines 1–3) that is used by each thread to identify their current shortest tour. Each thread begins by selecting a city at the position in the tour equal to their threadID (lines 5–8), resulting in a consecutive block of cities to be loaded into SHARED (line 10). Each thread then calculates the tour length change if they were to swap with the last city in the tour, then the next to last city, and so on, saving the current best possible swap seen (lines 12–17). After each thread has evaluated each pair of cities, the block of cities is shifted down to the next consecutive unseen cities (line 9). This process continues by moving the front cities until there are no more cities. The max(BEST) identifies the tour with the maximum decrease in the graph, in essence, the minimum tour length among all threads, and then the swap is performed (lines 21 and 22).

By organizing the computation in this way each thread can examine the exact same back city (denoted by the variable j in the for-loop starting in line 9) at the same time. This allows for shared

memory to be utilized as a cache that is pre-loaded with adjacent cities starting from the back of the tour. A broadcast of the data will get the next value for the back city to all threads efficiently. Once the back city has reached the front, the threads load the next city to use as the front. This will be *number_threads* away in the tour. It then repeats the process. After each pair of cities has been evaluated, we take the best swap from all the threads, apply that swap, and restart the process again with a new randomly generated tour. The process terminates when no new swap is found. The results are copied to the host from the GPU when all blocks have terminated.

3.2 Initial Code Modifications

We discovered that there was a scalability issue on the number of random tours attempted in the original 2-opt implementation. This is because each block only works with a single tour and each block must have memory allocated upfront. Since there is no memory allocation on the device side, all memory allocations were performed on the host side and addresses were passed in through the kernel call. For our first modification, we employ the use of C++ templates, which lets our program allocate shared memory in the kernel and not through the kernel call. In other words, we can use the template parameter to specify the exact amount of memory to allocate in the kernel. This can lead to a micro-optimization and increase performance if we continuously run the algorithm on multiple graphs. For the next modification, we pass pointers to shared memory with the restrict modifier, which informs the compiler that this pointer will not point to memory that overlaps with any other pointer. Therefore, we reduce any temporary variables, which would have been required if the compiler could not determine memory accesses that would not overlap.

3.3 Architectural Modifications

Our next step was to increase the sizes of the graphs allowable and increase the maximum number of feasible restarts by building on the code modification in the above section. We accomplish this using a fixed number of blocks and a work queue.

3.3.1 Fixed Number of Blocks. As mentioned in the previous section, one of the limits to the scalability of the original 2-opt implementation was due to memory allocation being performed from the host and not the device. The maximum number of restarts (R_{max}) was a linear relationship described by the equation

$$R_{max} = min(MaxBlockLimit, TotalMem/|G|),$$

where |G| is defined as the size of the graph G, *MaxBlockLimit* is the maximum number of blocks that can be scheduled and *TotalMem* is the amount of memory on the GPU. Once the memory allocation on the GPU was moved from the host side to the GPU, the new equation for the maximum number of restarts (R_{max}) is the following:

$$R_{max} = \begin{cases} MaxBlockLimit & (MaxActiveBlocks * |G|) \leq TotalMem \\ 0 & otherwise, \end{cases}$$

where *MaxActiveBlocks* is the maximum number of blocks that can be active on the GPU. Each block allocates and deallocates all memory when it starts running and when it has finished running. Note that a schedulable block will sleep until it is activated, and once a block starts a job it cannot stop until its job has finished. Taking into account the MaxActiveBlocks and the fact that only a small subset of running blocks can be active allows each block to allocate more memory.

If we have more blocks scheduled than can be run on the GPU, then memory must be allocated on the CPU for each block and grows linearly in terms of restarts. (Restarts are described in Section 2.2.) The new equation for R_{max} allows fixing the number of blocks to the GPU's block execution limit, which is the number of blocks that can be executed in parallel. In addition, memory is allocated only once for each block and all of the memory used is on the GPU. As a result,

there is no limit on the number of restarts and the constraints on the maximum size of the graph are relaxed with respect to the number of restarts. Since each block performs the same algorithm, we divide the restarts among the blocks and reuse the resources allocated. This insight allows us to have as many restarts as wanted, given there is enough memory to hold a single tour on a block and each launched block is given some number of restarts.

During our experiments, we noticed blocks being scheduled to run once and then terminating in the original 2-opt implementation. This resulted in a small performance degradation and could also result in hardware degradation. In our fixed block modification, each block will execute multiple restarts before it is terminated, removing this performance degradation.

3.3.2 Work Queue. With a fixed number of blocks, we needed a work scheduling algorithm and initially decided on a naive solution of dividing the work evenly among the blocks. In addition to the naive approach for a work scheduling algorithm, we also consider that of dynamically allocating work as needed using a work queue. In our fixed block modification, we gave each block an even number of tours, which leads to some blocks finishing before others. To ensure that we utilize the full resources of the GPU, we need each block busy, which is solved by using a work queue implemented on the GPU. We used a simple queue implementation that returns a single integer value representing the starting tour on which to run 2-opt. Our queue works by performing an atomic addition operation to a global integer variable to keep track of how many tours have been distributed to the blocks. Each block can see and modify this global variable, which is incremented by how many graphs are generated. If Q number of tours are to be scheduled per block, then the block calls a method that uses the Q as the starting number of the graphs that will be generated. Q is also used to determine the ending graph numbers, allowing the iteration through each graph and the generation of a random permutation of the vertices representing the graph. To determine the best strategy, we experimented with several work queue configurations for different values for Q, with the results presented in Section 5.4.

3.4 Algorithm Modification—K-swap and ∞-swap

Upon further examination of the algorithm, we determined that because each thread could possibly find a valid update to the tour, it could be possible to apply more than one update during the update phase. This led to our main contribution being the application of multiple updates to the tour found by each thread in the search phase. The update phase is modified to repeatably apply the best update that does not break the tour or overwrite a previous update. One of our concerns about the original algorithm was that we would find the same updates multiple times during the algorithm, leading to redundant work. This led to our modification to apply all updates in order of greatest improvement where they do not contradict previous updates.

In addition, due to the way the tour is stored, when some updates are applied they leave remaining candidate updates referencing positions in the tour that have moved. This is mitigated by discarding any update that overlaps the current best update with regard to the tour between the updated edges. As an example of an invalidated update, we consider the graph in Figure 4. Possible swaps involve the crossed edges (5,6) and (7,8) as well as (5,6) and (3,8) in the top graph. If the swap (5,6) and (7,8) is the better swap, then the update (5,6) and (3,8) becomes invalid, because the edge (5,6) is no longer in the graph. Hence, any update that matches this scenario must be ignored due to invalidation. Any update that is in between is also ignored due to invalidation. The bottom graph shows the graph resulting from the swap.

One of the expectations that arose was that placing a limit on the number of updates applied could lead to better performance. We tested different numbers of allowable updates per iteration and introduce the variable k, where k represents the number of swaps allowed. The final version

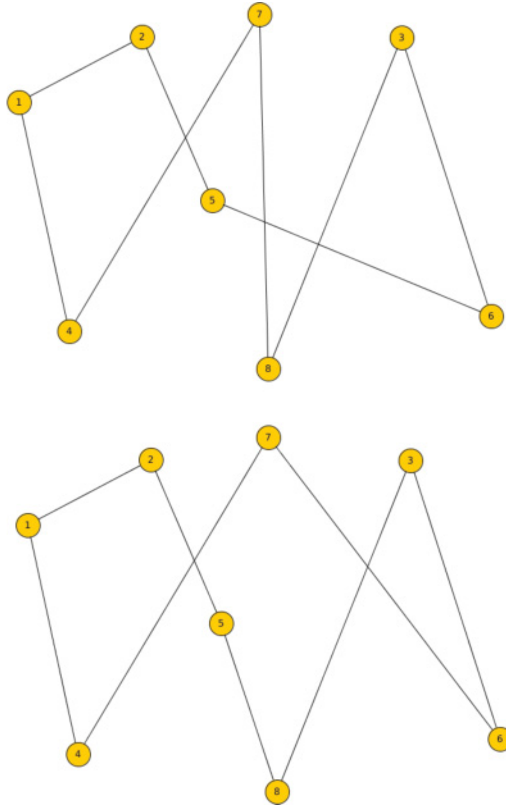


Fig. 4. Invalidation example.

for $s = 0$ to $k - 1$ {k number of swaps allowed} **do**
  $BEST_TMP = max(BEST)$
  $perform_swap(BEST_TMP)$
  if $BEST_TMP == BEST$ **then**
    $BEST_TMP = -1$
  end if
end for

Fig. 5. Pseudocode for k-swap implementation with the assumption of k available swaps.

of the modified algorithm is our *k-swap* algorithm, and if we allow unlimited updates, we call it ∞-swap. Figure 5 illustrates the changes to the original code in Figure 3 that are necessary to include the *k-swap*. Lines 21 and 22 of the original code are replaced by the code that finds the k best swaps instead of a single swap.

4 EXPERIMENTS

In this section, we discuss the goals of our article, introduce terms and definitions, and discuss our dataset and metrics. We also describe the types of experiments and experimental setting, and we conclude with threats to validity.

4.1 Definition and Context

Our goals are to show how the configuration and modification of an algorithm affect its performance and results. The configuration of the algorithm includes several parameters: the number of threads (**T**), the number of nodes (**N**), the amount of shared memory allocated (**S**), the number of tour updates per iteration allowed (**k**), the number of random tours (**R**), and the input data. The value of **T** is the number of threads that are within a block and **S** is the number of shared-memory elements, where each element is a triplet composed of the x and y coordinates of a node and the cost on the edge from that node to its neighbor in the tour. The values of **T** and **S** are from the set $\{32, \ldots, 32 * i, \ldots, 1024\}$. The values chosen for **k** are from the set $\{1, \ldots, 2^i, \ldots, 256\} \cup \{\infty\}$. In the text, we substitute the value ∞ for **k**, such that ∞-swap denotes an unlimited number of swaps.

In addition to the configurations above, we also examined how a work queue would affect performance. We tested a simple work queue that would give out a fixed amount of work (**Q**) to a block as requested. The value of **Q** can either be 1 tour, 100 tours, or **T**. When the value of **Q** is **T**, we say that the queue is disabled and the work is evenly divided among the blocks, e.g., no queue.

4.2 Data Sets

TSPLIB[2] is a collection of problem instances for the TSP [16] and has been used extensively in the TSP literature for benchmarking purposes. We are limited to using problems with the type *EUC_2D*, which allows for graphs where distances are Euclidean and lie in the x-y plane. The subset of problems we select are fl417, fl1400, fl3795, fnl4461, pla7397, usa13509, and pla33810. For each of the problems the numerical values appended represent the number of nodes in the graph and the prefix represents the type of problem.

4.3 Metrics

Previous work has used the number of 2-opt operations per second to record the performance. The modification of our algorithm significantly reduces the number of 2-opt operations per second but still generates equitable results, so that 2-opt moves per second is no longer a valid metric. We focus on other metrics, including duration and tour lengths, and quality of the tour lengths. Duration is recorded in milliseconds but is presented in seconds. The results are unitless and the quality of the result is the percentage of original implementation results.

4.4 Types of Experiments

Our hypothesis is that our modifications will improve the performance and the quality of the result. To reiterate, our modifications are the following: a reduced number of blocks, inclusion of a work queue, and incorporating *k-swap*. To determine the impact of these modifications on program performance and solution quality, we have designed the following experiments.

To study the affect of the *k-swap* value on the program performance and solution quality, we test different values of *k-swap* on six selected TSP problems and compare the results to the original implementation. In this experiment, we use a simple work queue. Each problem instance is given 1,000 random tours with the maximum number of threads and maximum amount of shared memory. We set the swap counts to powers of 2 ranging from 1 to 128. We also look at $k = \infty$. We do this for all data sets except pla33810, due to limitations in the original algorithm's implementation. The input file pla33810 has over 33,000 nodes and is run on a subset of the *k-swap* values, specifically, 1, 4, 64, 128, 256, and ∞-swap, due to time constraints.

[2] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.

To examine the effect of the thread and shared-memory configuration on the program performance and solution quality, we run our final version of our algorithm using the TSP problems fl1400, fl3795, and fl7397. These experiments are very time intensive, so for this experiment, we chose the largest three data sets that can produce results in a reasonable amount of time. We run each problem on multiple threads and shared-memory configurations, set *k-swap* equal to ∞-swap, and use the simple work queue. The number of threads and amount of shared-memory elements range from 32 to 1,024—generating a total of 1,024 results per TSP instance.

To compare the original version and the final version, we perform two experiments. In the first experiment we determine if there is any change in the quality of the solutions generated by the original algorithm compared to our modified algorithm. We run the original and final modified algorithms 1,000 times with 1,000 random tours each on the TSP problem fl1400 and fl3795. Because the original algorithm is extremely slow for many of the data sets, but ran in a reasonable amount of time for the fl1400 and fl3795, these data sets were chosen for a comparison in this experiment. The second experiment is used to determine the quality of the solutions we generate when both programs are given an identical amount of time. For this, we perform the same experiment above for the original algorithm, but we change the number of random tours to 8,255 for the final version. The purpose of this experiment is to compare the quality of solution when both programs are given the same amount of time.

Our final experiment examines multiple queue configurations and their impact on performance. In addition to different queue configurations, we also examined how the queue would be affected by hardware. In this experiment each run is given 10K, 100K, and 1M random tours, and a **Q** of 1, 100, or an equal amount of work. We also selected the number of blocks to be either 10K or the maximum resident blocks that the card can hold. For this experiment, we choose the problem fnl4461.

To decide on how many blocks to start, we use the CUDA API to obtain the exact number of active blocks that a card can support, which is fixed per the architecture. We choose between this number and the restart count, requiring allocation of memory once for each block instead of multiple times per random restart. We use the CUDA timers for measuring the execution time. For quality purposes, we use the Welch two sample t-test to compare the quality of our solutions to the original.

4.5 Setting

We have two machines on which we run our experiments. Each machine is running Windows 7 and uses CUDA Release version 6.5. The first machine has an Intel i3 with 16GB of system RAM and a GTX 780 with 3GB of GPU RAM. The second machine has an AMD Phenom II X4 965 Processor with 16GB of system RAM and has a GTX 650 Ti with 2GB of GPU RAM. During experiments all non-necessary services and programs were terminated and all video output was performed through an alternative GPU card. We used the 2013 Visual Studio compiler with the CUDA compiler version 6.5.10 with the flags:

$$-Xcompiler''-Ox'' - O3 - use_fast_math - lineinfo - -ptxas - options = -v$$
$$- m64 - gencodearch = compute_35, code = sm_35.$$

4.6 Threats to Validity

Because of the length of time to run all configurations on a problem set, we could not run them multiple times. This can lead to noise in the data, skewing our results and conclusions. While we did not run each configuration multiple times to smooth out noise, we did identify patterns that occur between different problem instances.

In addition to multiple runs per configuration, we could have tested larger datasets. This was also not feasible, as we estimated the time to compute the next largest dataset, usa13509, would take approximately a month to complete for testing all configurations on our setup.

5 RESULTS

In this section, we first present our main contribution, *k-swap*, and how different values affect the generated solutions and performance. The second section presents results pertaining to how the runtime configuration of our algorithm affects the performance and solutions. In these experiments, each thread configuration is in terms of how many threads are in each block and the shared-memory configuration is in terms of nodes, where each node holds all information required to make a decision about a city for swapping. Then, we look at how the quality of our algorithm compares with the initial algorithm. Finally, we talk about how a work queue affects the duration of the algorithm. All the results in Sections 5.1–5.3 were generated using the GTX 780 GPU, while both the GTX 780 and 650Ti were used in Section 5.4.

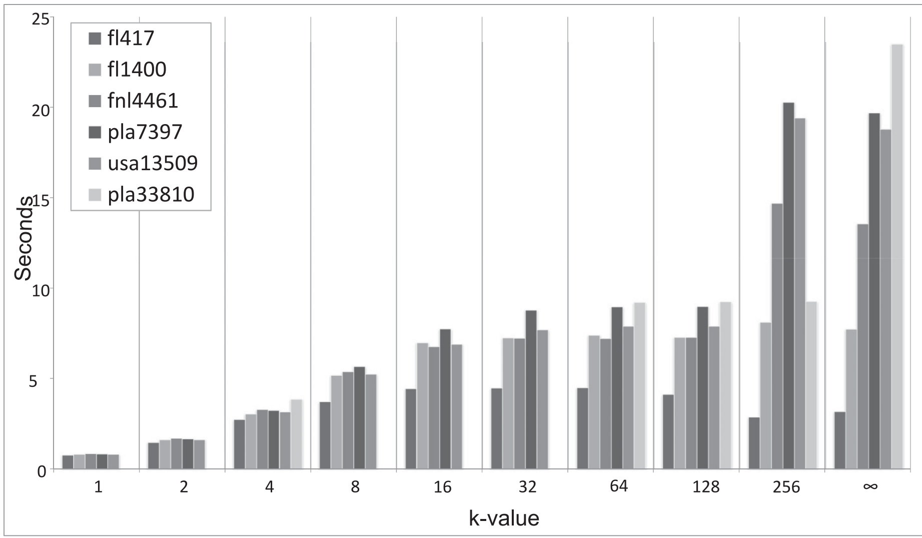
5.1 Effects of K-swap

Figure 6(a) illustrates the speed-up provided by *k-swap* compared to the original 2-opt algorithm as the number of values of k varies from 1 to ∞. In general, we found that increasing the number of allowed updates after the searching phase increased performance significantly, as shown in Figure 6(a). However, some problem instances did not keep increasing as the number of allowed swaps increased, but reached a maximum performance gain beforehand. An example is pla7397, which reaches its peak at 256 allowed swaps.

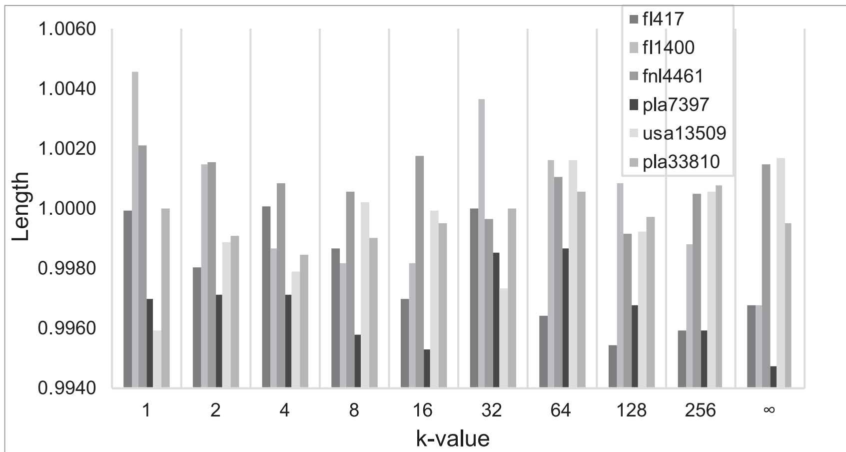
For problem pla33810, the original implementation cannot run the graph due to constraints in the original programming, which places a limit on size. Therefore, speedups are compared to the runtime when we only allow a single swap. For all other problems, there is a performance drop with our version when compared to the original when a single allowed update is allowed. For all other swap values, we achieve a speedup. We found that our speedups range from 1.48 to 23.48 with smaller graphs achieving smaller overall speedups, while larger graphs achieve much higher speedups.

To see how the *k-swap* value affects the tours found, we recorded the best tour lengths found, and we present them in Figure 6(b). These results show the ratio of the tour found using the given *k-swap* value against the tour length found by the original implementation for each problem instance. As before, we could not run pla33810 on the original version due to limitations on tour sizes, so the values presented occur when the *k-swap* value is one. We found that the tour ratio values range from 0.9947 to 1.0046 with an average of 0.9992 and standard deviation of 0.0022. While most tour lengths found by *k-swap* were shorter than those found by the original implementation, this is not true of all tour lengths, as shown in Figure 6(b). However, because we are able to find a solution in faster time with *k-swap*, we could potentially run *k-swap* with even more random restarts and get even shorter tours. This will be explored in our future work.

From the results shown in Figure 6(a), we see that increasing the *k-swap* values provides a large performance gain, with larger gains for larger problems. However, for smaller problems, we see that after a certain threshold the amount of gains begins to decrease. This is because as the graph gets smaller, the probability of updates being invalidated increases. This result also implies that as the graph gets smaller and the *k-swap* value increases the larger improvements are discarded due to invalidation. (A discussion of invalidation appears in Section 3.4.) The current theoretical speedup is 1,024, because each block is allowed to only have 1,024 threads as each thread only handles a single update. Allowing for threads to keep a list of top updates they have seen can expand this theoretical bound even higher. Additionally, this approach will only work for sufficiently large



(a) Speedup provided by k-swap



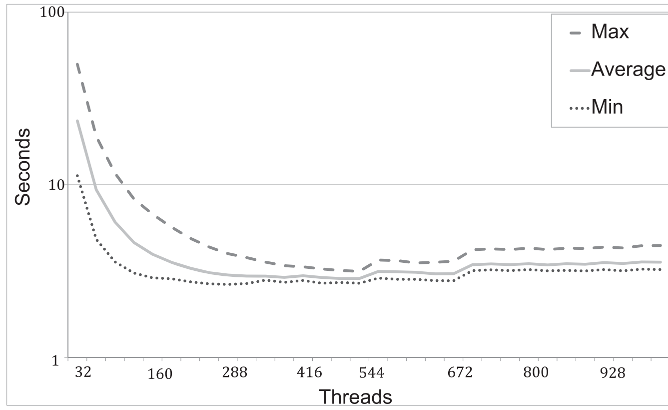
(b) k-swap effect on tour lengths

Fig. 6. Given different k-values, we present speedup 6(a) and proportion of tour lengths of our algorithm compared to original 6(b).

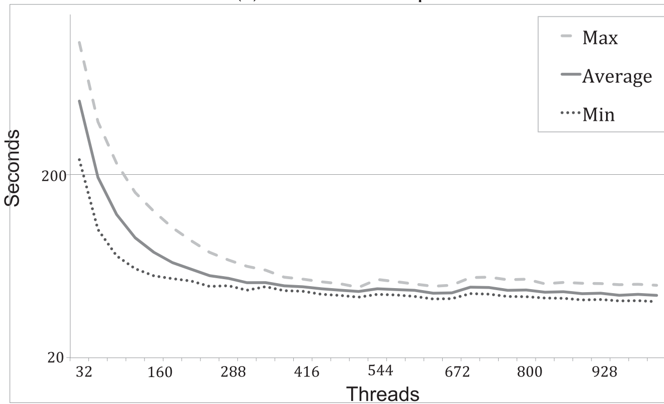
problem instances due to the overhead required to keep track of the top updates and then apply them.

5.2 Effects of Configuration: Results

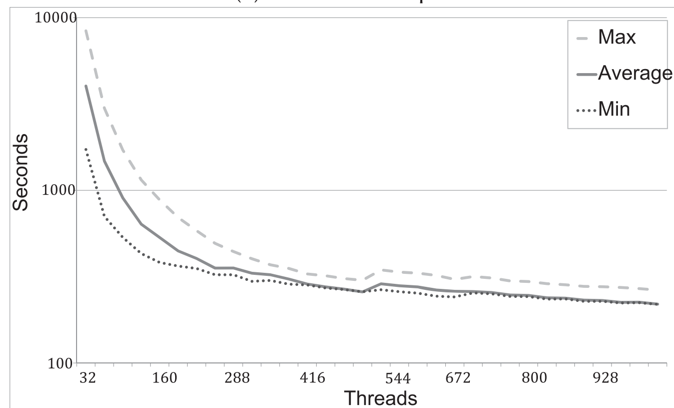
5.2.1 Runtime Per Thread. We expect that having more threads results in better performance, because the amount of memory fetches is proportional to the number of threads, $fetches \in O(N^2/T + N)$. The minimum, maximum, and average runtimes as the number of threads increases, appear in Figures 7(a), 7(b), and 7(c), for graphs with 1,400, 3,795, and 7,397 nodes, respectively. In Figure 7, the x-axis corresponds to the number of threads per block and the y-axis is a log scale



(a) 1400 Node Graph.



(b) 3795 Node Graph.



(c) 7397 Node Graph.

Fig. 7. The minimum, average, and worst case runtime per thread configuration. The x-axis corresponds to the number of threads per block and the y-axis is a log scale with units of seconds representing the duration of the algorithm.

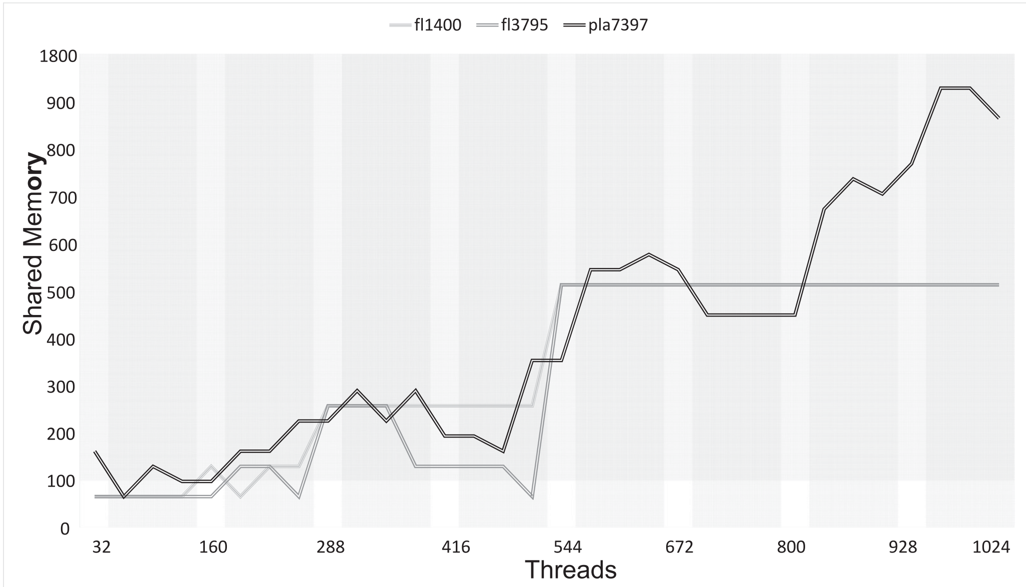
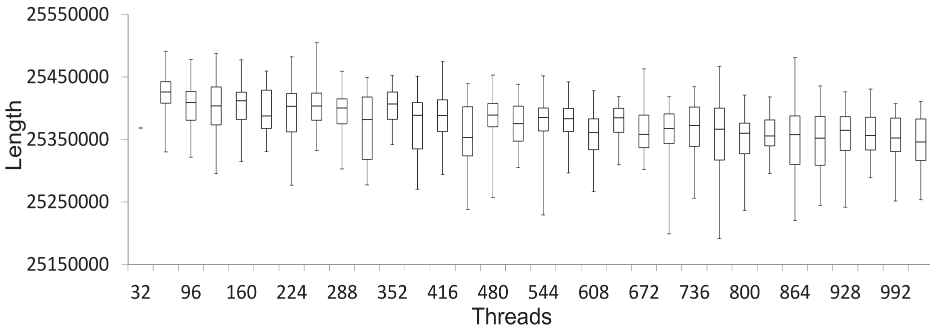


Fig. 8. Shared-memory configuration per thread count configuration which results in the best runtime performance.

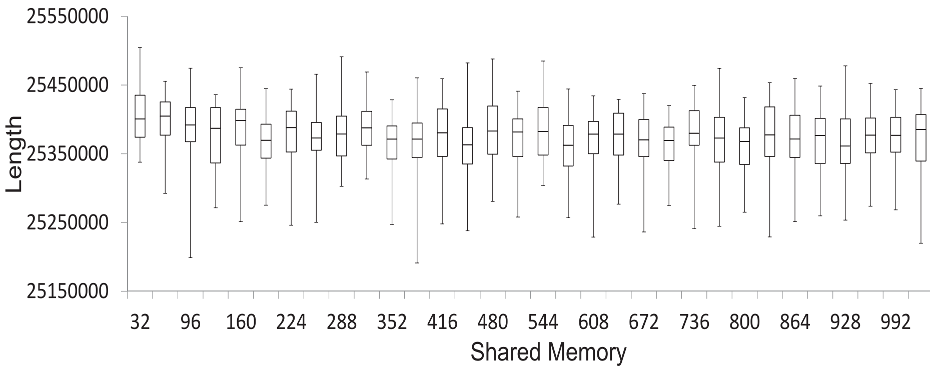
with units of seconds representing the duration of the algorithm. We found that the worst performance occurs when using 32 threads for all problem instances. The two larger problem instances (3,795 and 7,397 nodes) achieve the best performance with 1,024 threads and the smaller problem instance (1,400 nodes) achieves the best performance with 288 threads as shown in Figure 7. We also see that for all problem instances, an increase from 512 to 544 and 672 to 704 threads resulted in a performance drop, with smaller reductions as the problem instance size increases. This drop is caused by a reduction in the number of active blocks. Because only 2,048 threads can be allocated per block, we have $\lfloor \frac{2,048}{threads} \rfloor$ blocks active, leaving threads underutilized for some thread sizes.

We also see from the graphs that our assumption that more threads would lead to better performance is validated for the larger problem instances. For the smaller problem instance, we find that performance peaks at 288 threads and then periodically remains stable, with performance decreasing as we add more threads. This is because the amount of work performed in the search phase is $O(N^2/T + N)$ on the number of vertices and the update phase is $O(Tlog(T) + N/T)$ on the number of threads, with a large constant due to a high density of synchronization calls. This means for smaller problems with more threads the search phase can be dominated by the update phase where, as before, the update phase was constant. In addition, as the size of the graph grows the probability of two swaps overlapping decreases, and as the number of threads increases the probability of invalidation increases. This implies that more speedup can be attained for larger graphs with more threads. However, for smaller graphs with more threads the amount of invalidations increase. This makes the overhead of invalidation resulting in no speedup or negative speedup.

5.2.2 *Shared Memory Per Thread.* Figure 8 presents the shared-memory configuration (number of memory elements) for which a given thread configuration resulted in the best performance for the problem instances fl1400, fl3795, and pla7397. For the two smaller problem instances, the shared-memory values range from 64 to 512 elements, and after 512 threads the shared-memory



(a) Length statistics per thread count configuration



(b) Length statistics per shared memory configuration

Fig. 9. Box-plots for pla7397 which give the minimum, first-quartile, median, third-quartile, and maximum lengths found for each configuration value.

configuration plateaus at 512 for the best performance. The largest problem instance has shared-memory values ranging from 64 to 1,024 for the best performance and has an overall increasing trend as the number of threads increases. We expected that the shared-memory and thread values for the best performance would be increasing due to the fact that as the ratio of shared memory to thread increases, the number of active blocks allowed decreases, similar to the curve for pla7397. For the other two problem instances, we believe that the amount of work per thread and shared memory per thread are the limiting factors.

5.2.3 Length Per Thread and Shared Memory. In addition to performance, we also examined the solutions (tour length) generated by different configurations of thread count and shared memory for fl1400, fl3795, and pla7397. For shared memory, the tour length median values do not appear to have any correlation with the configuration. In Figure 9, we illustrate for pla7397 the results for thread count and shared memory as histograms in Figures 9(a) and 9(b), respectively. At first glance, there appears to be a slight correlation between the thread configuration and the median values for pla7397 in Figure 9(a) and also for fl3795, although not shown here. We did not observe any such correlation for fl1400. We also found that for pla7397 with a thread configuration of 32, we always generate the same solution and for fl1400 the runtime configuration of 640 threads with 32 shared-memory elements generates an outlier with a tour length of 20,108 consistently.

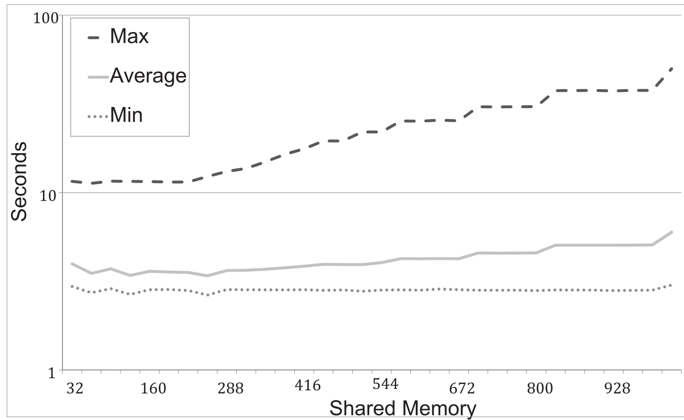
Table 1. Hypothesis Tests with Alpha-level of 0.05 and Null-hypothesis *"The Correlation Between Configuration Value and Length Is Non-negative"*

Configuration	Problem	Correlation	P-value	Decision
Shared memory	fl1400	−0.1265	0.2467	Retain
	fl3795	−0.4159	0.0086	Reject
	pla7397	−5380	0.0006	Reject
Thread count	fl1400	−0.3035	0.0457	Reject
	fl3795	−0.8332	5.4E-11	Reject
	pla7397	−0.8234	1.6E-10	Reject

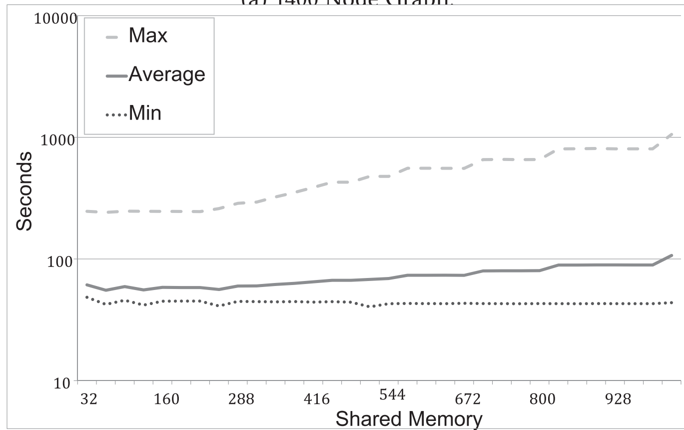
5.2.4 Solution Quality and Configuration. Due to how graphs are generated and how the algorithm works, we did not expect any correlation between solution quality and configuration. To see if the configurations had any affect on the solution quality, we used Fisher's z-transform with a null-hypothesis that *the correlation between configuration value and tour length is non-negative.* Using an α-level of 0.05, we found that we reject the null-hypothesis in all cases except for the shared-memory configuration as shown in Table 1. We believe that the test fails to reject the null-hypothesis for fl1400 because of the extreme outlier at 32 shared-memory elements, and without this outlier the result would have been a rejection.

5.2.5 Runtime Per Shared Memory. For all shared-memory configurations the best runtime appears to be independent of the shared-memory values as shown in Figure 10. However, the worst runtime monotonically increases as the shared-memory configuration increases. In Figure 10 the x-axis is the amount of shared memory allocated in terms of information stored by each node in the graph, and the y-axis is a log scale with units of seconds representing the duration of the algorithm. This figure illustrates the following: for a given amount of shared memory, how many threads give optimal performance. The average runtime for each problem instance begins around 7−12% of the distance between min and max and quickly reduces to 6% of the distance between the two. The worst min runtime occurs at 32 threads for all problem instances, and the curves for the best duration are centered around 288, 960, and 1,024 threads for problem instances fl1400, fl3795, and pla7397, respectively. The worst min runtime occurs at 32 threads because of three reasons: the runtime of the search phase is $O(N^2/T + N)$, shared memory is limited, and 128 threads is the largest number of threads before the number of active blocks decreases. Because the search phase speedup is determined by the number of threads, this means fewer threads result in worse performance for large enough graphs. Because shared memory is limited, we can only allocate at most 3,072 bytes per block, or 256 elements, to achieve the maximum active block amount with 32 threads. With 1,024 elements per block out of 48KB of shared memory, we can have at most 4 active blocks per SMX, resulting in 128 active threads. From this, we can state that no matter what shared-memory configuration we choose, we can always find a thread configuration that results in good performance.

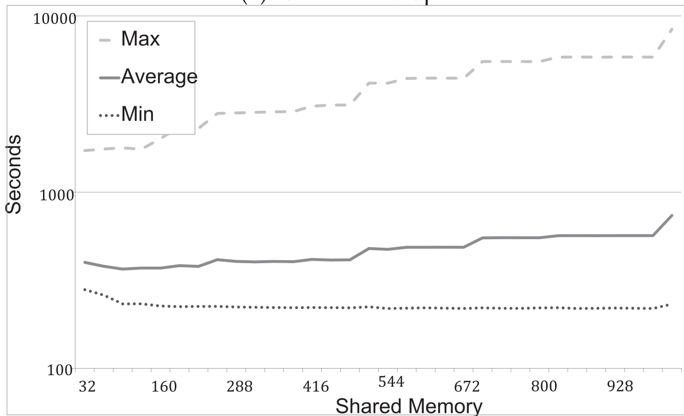
5.2.6 Thread Count Per Shared Memory. In Figure 11, we show the shared-memory configuration, given the specific thread value, which resulted in the best runtime for the 1,400, 3,795, and 7,397 node graphs. For fl3795 and pla7397, we see that the best performance occurs when the thread configuration is at or near 1,024 threads for most shared-memory configurations; however,



(a) 1400 Node Graph.



(b) 3795 Node Graph.



(c) 7397 Node Graph.

Fig. 10. The minimum, average, and worst case runtime per shared-memory configuration. The x-axis is the amount of shared memory allocated in terms of information stored by each node in the graph and the y-axis is a log scale with units of seconds representing the duration of the algorithm.

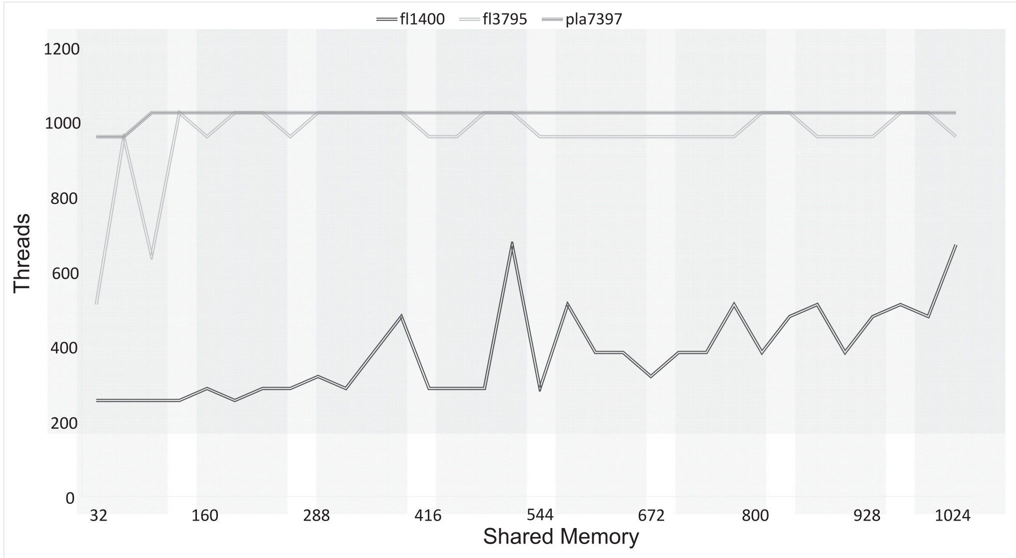
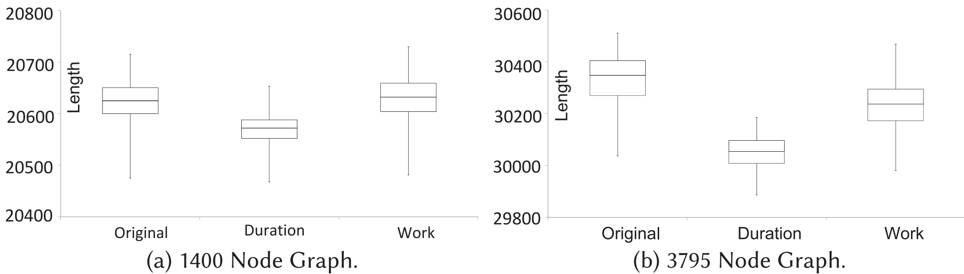


Fig. 11. Best thread count configuration per shared memory configuration.



(a) 1400 Node Graph. (b) 3795 Node Graph.

Fig. 12. Distribution of lengths with the original graph or with the modified version given equal time or equal work. Panel (a) presents results for fl1400 and panel (b) for fl3795.

for fl1400 the result is more erratic. As mentioned previously, we expected more threads to lead to better performance, which the curves for fl3795 and pla7397 illustrate.

5.3 Comparison of Solution Quality

As we expected, as shown in Figure 12(a), giving our k-swap algorithm the same amount of time as the original version of the algorithm resulted in shorter tour lengths for both graphs fl1400 and fl3795. The *k-swap* algorithm also provides shorter tour lengths when the same amount of work, in terms of number of restarts, is given to both algorithms for the graph with 3,795 nodes as shown in Figure 12(b). However, with the same amount of work, the original version results in slightly shorter tours for the smaller graph fl1400. We find that in our experiment *k-swap* provides the most advantage for the larger graphs. In future work, we would like to further study our implementation with larger problem sets. Unfortunately, we estimate the experiment with the next larger problem instance would take four months to complete with the current experimental configuration.

Table 2. Results of the Queue With the Number of Blocks Limited to 10K Blocks with Problem inStance fnl4461

Restarts	Time w/ 1	Time w/ 100	No Queue
10,000	2,186	1,982	2,186
100,000	19,665	19,664	21,627
1,000,000	197,704	197,704	218,752

(a)GTX 650 Ti with 10K blocks

Restarts	Time w/ 1	Time w/ 100	No Queue
10,000	707	787	689
100,000	7,068	7,126	6,887
1,000,000	70,688	70,761	68,886

(b)GTX 780 with 10K blocks

Table 3. Results of the Queue with the Number of Blocks Limited to the Maximum Resident Blocks per Card With Problem Instance fnl4461

Restarts	Time w/ 1	Time w/ 100	No Queue
10,000	1963	1192	2178
100,000	19653	19662	21853
1,000,000	196564	196564	218711

(a)GTX 650Ti with 64 blocks

Restarts	Time w/ 1	Time w/ 100	No Queue
10,000	707	787	689
100,000	7069	7126	6886
1,000,000	70688	70749	68886

(b)GTX 780 with 192 blocks

5.4 Effects of a Work Queue

As shown in Tables 2(a) and 3(a), there is a positive linear correlation between the duration and the number of restarts. For the GTX 650, the best performance occurs with a work queue. The runtimes are shorter with a work queue than without a work queue for the GTX 650, with the longer queue providing a larger improvement for fewer restarts. We see that the duration differences between a queue of size 1 and a queue of size 100 becomes negligible as the number of restarts increases. Our results show that scheduling fewer blocks does not negatively affect performance as expected. Given either 10K blocks or the maximum hardware limit for active blocks, we achieve similar runtimes for the same work queue and tour count configurations.

In our experiments, however, the work queue hurts performance for the GTX 780. As shown in Tables 2(b) and 3(b), having no queue for the GTX 780 results in better performance for each different number of random tours. We believe that the difference between the two GPUs is due to the architecture and hardware. The GTX 780 has more memory bandwidth and a higher clock frequency than the GTX 650, and in addition the compute architecture is 3.5 versus 3.0. The new compute capability of the 780 provides increased computational performance and thread memory instructions that allow for threads in a warp to share data. We believe that this improvement in performance for computation puts a bottleneck on expensive memory operations such as atomic memory operations.

6 CONCLUSION

The Traveling Salesman Problem is an important NP-hard problem connected to many real-world issues. As of this writing and for the foreseeable future, there does not exist any polynomial time algorithm to solve all instances of the TSP. Because of this, we are left with approximation algorithms, of which an iterative hill-climbing method using 2-opt is one. In this article, we presented our k-swap modification of an existing iterative hill-climbing 2-opt algorithm implemented for the GPU. We have shown that with our modifications there is a substantial speedup without a reduction in the quality of the result by applying the *k-swap* method. Our results showed that

common assumptions on how to get good performance for the GPU are not always true, such as saturating the GPU with blocks. Instead, for problems in which one can deterministically enumerate the search space, the number of active blocks can be limited as determined by the hardware. This property allows for reduced memory allocation. Since each block can allocate the amount of memory upfront, the result is the ability to use a limited amount of memory. For our problem, and those that have a similar layout in which we perform some global search and apply an update, we present configuration values that result in maximal performance.

7 FUTURE WORK

In a future version, we would like for multiple swaps to happen in parallel and to determine a way to reduce the number of swaps we have to invalidate. Future work could also consider more advanced k-opt versions such as 3-opt and k-opt and determine what limitations the GPU presents for those algorithms. One current limitation we have now is that if a graph is too large, then we might not be able fill the GPU with blocks. To use all the resources, we would have to distribute a single tour across multiple blocks, requiring interblock communication, which would be beneficial for many other problems. We also believe a more in-depth study of work queues would be beneficial not only for our problem but other work as well. From a theoretical standpoint, we would like to see bounds on the runtime as well as the efficiency of our algorithm.

REFERENCES

[1] David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. 2007. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics).* Princeton University Press, Princeton, NJ.

[2] Peter Bakkum and Kevin Skadron. 2010. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU'10).* ACM, New York, NY, 94–103.

[3] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. 2004. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. Graph.* 23, 3 (Aug. 2004), 777–786.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press.

[5] A. Croes. 1958. A method for solving traveling salesman problems. *Operat. Res.* 5 (1958), 791–812.

[6] Matthias Englert, Heiko Röglin, and Berthold Vöcking. 2014. Worst case and probabilistic analysis of the 2-Opt algorithm for the TSP. *Algorithmica* 68, 1 (2014), 190–264. DOI: http://dx.doi.org/10.1007/s00453-013-9801-4

[7] David S. Johnson and Lyle A. Mcgeoch. 1997. *The Traveling Salesman Problem: A Case Study in Local Optimization.* John Wiley and Sons, London.

[8] Tonci Caric Juraj Fosin, Davor Davidovic. 2013. A GPU implementation of local search operators for symmetric travelling salesman problem. *PROMIT—Traffic and Transportation* 25, 3 (2013), 225–234.

[9] E. Scott Larsen and David McAllister. 2001. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC'01).* ACM, New York, NY, 55–55.

[10] Marek Libura, Edo S. van der Poort, Gerard Sierksma, and Jack A. A. van der Veen. 1998. Stability aspects of the traveling salesman problem based on k-best solutions. *Discrete Appl. Math.* 87, 13 (1998), 159–185.

[11] Shen Lin and Brian W. Kernighan. 1973. An effective heuristic algorithm for the travelling-salesman problem. *Operat. Res.* 21 (1973), 498–516.

[12] Michael McCool and Stefanus Du Toit. 2004. *Metaprogramming GPUs with Sh.* AK Peters Ltd.

[13] NVIDIA Corporation. 2007. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide.* NVIDIA Corporation.

[14] Molly A. O'Neil and Martin Burtscher. 2015. Rethinking the parallelization of random-restart hill climbing: A case study in optimizing a 2-opt TSP solver for GPU execution. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs (GPGPU'15).* ACM, New York, NY, 99–108. DOI: http://dx.doi.org/10.1145/2716282.2716287

[15] Molly A. O'neil, Dan Tamir, and Martin Burtscher. 2011. A parallel GPU version of the traveling salesman problem. In *2011 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11).* Las Vegas, NV.

[16] Gerhard Reinelt. 1991. TSPLIB—Traveling salesman problem library. *ORSA J. Comput.* 3, 4 (21 Nov. 1991), 376–384. DOI: http://dx.doi.org/10.1287/ijoc.3.4.376

[17] Kamil Rocki and Reiji Suda. 2013. High-performance GPU accelerated local optimization in TSP. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW'13)*. IEEE Computer Society, Washington, DC, 1788–1796.

[18] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test* 12, 3 (May 2010), 66–73. DOI: http://dx.doi.org/10.1109/MCSE.2010.69

[19] D. Strigl, K. Kofler, and S. Podlipnig. 2010. Performance and scalability of GPU-based convolutional neural networks. In *Proceedings of the 2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'10)*. 317–324.

[20] Akihiro Uchida, Yasuaki Ito, and Koji Nakano. 2012. An efficient GPU implementation of ant colony optimization for the traveling salesman problem. In *Proceedings of the 2013 International Conference on Computing, Networking and Communications (ICNC'12)*, 94–102.

[21] M. G. A. Verhoeven, E. H. L. Aarts, and P. C. J. Swinkels. 1995. A parallel 2-opt algorithm for the traveling salesman problem. *Future Gener. Comput. Syst.* 11, 2 (March 1995), 175–182.

[22] Hongjian Wang, Naiyu Zhang, and Jean-Charles Crput. 2013. A massive parallel cellular GPU implementation of neural network to large scale euclidean TSP. In *Advances in Soft Computing and Its Applications*, Felix Castro, Alexander Gelbukh, and Miguel Gonzalez (Eds.). Lecture Notes in Computer Science, Vol. 8266. Springer, Berlin, 118–129.

Received March 2017; revised August 2017; accepted October 2017