Rethinking the Parallelization of Random-Restart Hill Climbing

A Case Study in Optimizing a 2-Opt TSP Solver for GPU Execution

Molly A. O'Neil
Department of Computer Science
Texas State University
San Marcos, TX USA
moneil@txstate.edu

Martin Burtscher
Department of Computer Science
Texas State University
San Marcos, TX USA
burtscher@txstate.edu

ABSTRACT

Random-restart hill climbing is a common approach to combinatorial optimization problems such as the traveling salesman problem (TSP). We present and evaluate an implementation of random-restart hill climbing with 2-opt local search applied to TSP. Our implementation is capable of addressing large problem sizes at high throughput. It is based on the key insight that the GPU's hierarchical hardware parallelism is best exploited with a hierarchical implementation strategy, where independent climbs are parallelized between blocks and the 2-opt evaluations are parallelized across the threads within a block. We analyze the performance impact of this and other optimizations on our heuristic TSP solver and compare its performance to existing GPU-based 2-opt TSP solvers as well as a parallel CPU implementation. Our code outperforms the existing implementations by up to 3X, evaluating up to 60 billion 2-opt moves per second on a single K40 GPU. It also outperforms an OpenMP implementation run on 20 CPU cores by up to 8X.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—Parallel Programming

General Terms

Algorithms, Performance

Keywords

program parallelization, code optimization, hill climbing, iterative local search, TSP, GPGPU, CUDA

1. INTRODUCTION

Multi-start local search algorithms are a common technique to address combinatorial optimization problems (COPs). Finding the optimal solution to these problems is often NP-hard, and heuristic algorithms that find near-optimal solutions are applied instead. Local search algorithms generate an initial candidate solution and then iteratively improve the solution through successive moves to neighbor solutions. In iterative hill climbing (IHC), the algorithm moves in each step to the neighbor that maximizes the outcome until it reaches a locally optimal solution, i.e., one that cannot be further improved by a move to a neighboring solution. Because local search techniques such as IHC will become stuck at local optima, multi-start search algorithms repeatedly apply local search from different initial candidate solutions [17]. A simple multi-start algorithm is random-restart hill climbing, in which IHC is repeatedly performed from random initial solutions and a global best solution tracked across all restarts. This process of local improvements and random restarts continues until the solution is sufficiently good or a limit on computing resources is reached [22]. Random-restart IHC can require thousands if not millions of restarts to produce a good solution with high probability, making it a computationally expensive approach.

The traveling salesman problem (TSP) [18] is a frequently explored COP that involves finding a minimum-distance Hamiltonian tour in a complete, undirected, weighted graph in which the vertices represent cities and the edge weights represent the distances between cities. The tour must traverse each vertex in the graph exactly once and end at the starting vertex. The optimal TSP solution consists of the Hamiltonian tour that yields the minimum distance traveled. Because it is simple to explain yet finding its globally optimal solution is NP-hard [14], TSP is often used as an early exploration ground for heuristic approaches to combinatorial optimization [15]. It is also an important problem on its own, with applications in domains such as logistics [11], wire routing [16], and genome analysis [1].

In the random-restart IHC approach to TSP, each iterated hill *climb* begins from a randomly-generated candidate solution, or *tour*. In each IHC step of the climb, a set of tour modifications, called *moves*, are evaluated to determine the best move (i.e., the move to the neighbor solution with the minimum tour length out of all available neighbor solutions). For instance, the tour can be adjusted by a heuristic such as 2-opt [6], which removes two edges of the tour, thus splitting the tour into two subtours, and then reconnects the subtours in the other possible direction preserving a legal tour. For example, Figure 1 illustrates a 2-opt move that

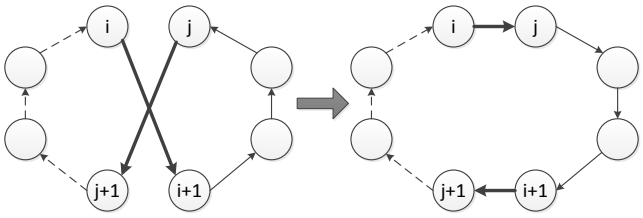

Figure 1. Illustration of a 2-opt move that replaces the bolded edges in the tour on the left with the bolded edges in the tour on the right. Note that the order of the tour portion on the right between the swapped edges is reversed.

GPGPU-8, February 07 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3407-5/15/02…$15.00
http://dx.doi.org/10.1145/2716282.2716287

removes the tour edges (v_i, v_{i+1}) and (v_j, v_{j+1}) and adds edges (v_i, v_j) and (v_{i+1}, v_{j+1}). At each IHC step, all possible 2-opt moves are evaluated and, if a move that reduces the tour length is found, the tour is modified by applying the best 2-opt move. The IHC iteration then continues by evaluating all possible 2-opt moves from the new tour; this process repeats until a local optimum is found (i.e., when no 2-opt moves can further improve the tour). At that point, the best tour and tour length are recorded and the process restarts from a new randomized candidate solution. After each climb, the local optimum is compared to the best solution thus far and saved if a new global minimum tour length has been found.

Because it is computationally expensive, exhibits large amounts of parallelism, and requires little synchronization, random-restart TSP is a promising candidate for general-purpose GPU acceleration. Within an individual hill climb, each IHC step depends on the previous step and therefore must execute serially. However, each of the restarts, or *climbers*, is independent and can be processed in any order, including concurrently. Within each IHC step, the evaluation of each possible 2-opt move from a tour is also independent. Two previous implementations of 2-opt TSP solvers that run entirely on the GPU exist in the literature. Both parallelize the climbers, with a climber assigned per thread and sequential 2-opt evaluation within each climber.

This approach has several disadvantages. In order to fully utilize the GPU, it requires a large number of climbers even when fewer climbers are sufficient to obtain good solution quality. Each climber must store its own tour order, resulting in a large memory footprint and reduced locality. Each climber executes its hill climb sequentially, including sequential move evaluation, which can lead to long latency to the discovery of the first local optima. Lastly, assigning each thread an independent climber results in branch divergence because nearby threads execute unrelated climbs. We present a 2-opt TSP solver that addresses these issues by assigning independent climbers to thread blocks rather than threads, utilizing the threads within a block to parallelize the move evaluations. This approach minimizes divergence, allows for more locality within each GPU core, greatly reduces the active working set size, and can fully load the GPU with fewer climbers.

This paper makes the following main contributions.

- We present a CUDA-based 2-opt TSP solver for large problem sizes that outperforms prior implementations.
- We describe several key optimizations included in our implementation and analyze their performance impact.
- We demonstrate that a hierarchical strategy for parallelizing climbers across blocks and move evaluation between the threads within a block may be the most effective approach to accelerating random-restart local search on GPUs.

The rest of this paper is organized as follows. Section 2 reviews existing work. Section 3 details our implementation and optimizations. Section 4 describes the methodology by which we measure the performance. Section 5 summarizes the individual GPU and CPU implementations we compare. Section 6 presents our findings and compares our implementation with prior work. Section 7 summarizes and concludes.

2. BACKGROUND AND RELATED LITERATURE

NVIDIA GPGPU architectures have a two-level compute hierarchy composed of multiple streaming multiprocessors (SMs), or *cores*, each composed of multiple tightly coupled *processing elements* (PEs). CUDA programs specify the behavior of parallel threads; these threads are grouped into blocks and the blocks are assigned to SMs as SMs become available. The threads within a block share a software-controlled cache (*shared memory*) and fast synchronization hardware. Between blocks, synchronization and data exchange require accesses through slower, off-core global memory. Sets of adjacent threads (called a *warp*, currently 32 threads for NVIDIA GPUs) within a block must share the same control-flow path to execute in parallel on the PEs. If they suffer from branch *divergence*, their execution will be serialized into smaller sets of threads with identical control flow. Additionally, memory accesses from threads within the same warp must be *coalesced* to the same 128B cache line or their execution will be serialized by the hardware. In general, GPUs require large amounts of parallelism and minimal global synchronization for good performance.

The majority of GPU-based approaches to the traveling salesman problem implement a variation of Ant Colony Optimization (ACO) [9], a meta-heuristic algorithm based on the natural ability of ants to discover, collaboratively, the shortest path between their nest and a food source by depositing pheromones along their traveled paths. ACO algorithms simulate the behavior of individual ants, which travel independently and construct a solution based on the pheromone trails left by other ants in previous iterations. Thus, this algorithm is highly parallelizable [10]. Many CUDA implementations of ACO TSP solvers exist [2][4][8][27]. ACO algorithms, while successful at solving many COPs, require the storage of a pheromone matrix that becomes impractically vast at large problem sizes [8]. There have also been a handful of GPU implementations of genetic algorithms (GAs) applied to TSP [5][13]. In this paper, we examine TSP as a test case for iterative hill climbing, which has many other applications (e.g., finding a maximal parsimony phylogenetic tree [12]). As such, we do not directly compare our 2-opt implementation to TSP solvers based on other algorithms.

2.1 2-Opt TSP for the GPU

Two previous publications present 2-opt TSP solvers that run entirely on the GPU.

O'Neil et al. [19] present a CUDA TSP solver capable of solving (with near-optimal solution quality) problem sizes up to 110 cities, achieving a speedup of 61X over a sequential CPU implementation and roughly equaling the performance of a pthreads implementation on 32 CPUs with 8 cores each. Their implementation parallelizes 100,000 random restarts by assigning individual climbers to threads. Initially, a single climber is assigned to each thread; successive climbers are obtained from a global worklist (via atomic increments) to ensure load balance between climbs, since climbs require a variable number of IHC steps to find a local minimum. A single critical section at the end of each climb tracks the global best solution via atomic operations on global memory.

Each thread serially performs the steps of the IHC algorithm, including sequential evaluation of all possible 2-opt moves at each step. The 2-opt moves are evaluated in two nested *for* loops that

```
// city[i] is the coordinate index of ith city
// in tour order
#define dist(a,b) dmat[city[a]][city[b]]
do {
  minchange = 0
  for (i = 0; i < cities-2; i++) {
    for (j = i+2; j < cities; j++) {
      change = dist(i,j) + dist(i+1,j+1)
               - dist(i,i+1) - dist(j,j+1)
      if (minchange > change) {
        minchange = change
        mini = i, minj = j
      }
    }
  }
  if (minchange < 0)
    // apply best 2-opt move
} while (minchange < 0)
```

Listing 1. Pseudo code for the nested loops that evaluate the 2-opt moves

compute the best pair of edges to remove and reconnect in the other direction. These loops are optimized to avoid duplication in city pairs due to symmetry as well as to prevent the evaluation of adjacent edges in the tour, the swapping of which can never result in a change of the tour length. For example, for a problem with n cities, the outer i-loop iterates from the 1st to the n-2nd city while the inner j-loop iterates from the i+2nd to the n^{th} city. The tour cost is never actually computed during the move evaluation; the change in tour cost of each move is sufficient to pick the move that results in the greatest tour length reduction. Pseudo code for the nested *for* loops that perform the move evaluation is shown in Listing 1. Additionally, the code optimizes the loop nest by register-caching the j-loop-invariant variables in the outer i-loop.

The code keeps an array of cities representing the tour order in local memory for each thread, which ensures cached and coalesced accesses. To allow for fast 2-opt evaluation, the distances between cities are pre-computed and stored in a two-dimensional distance matrix in shared memory. For a GPU with 48kB of shared memory, this $O(n^2)$ storage requirement limits the implementation to problem sizes of 110 cities or fewer.

Rocki and Suda [24] present a modification of the above implementation that allows for problem sizes up to around 6,000 cities. Their implementation is based on the insight that the computation-oriented architecture of GPUs encourages recalculating data rather than storing it in shared memory (which is very limited) or off-chip (which has very high latency). Instead of pre-computing the distance matrix and keeping it in shared memory, their code stores the city coordinates in shared memory and recomputes the distances between cities as needed. This implementation underperforms the distance matrix implementation by up to fifty percent; however, it has an $O(n)$ storage requirement and can solve much larger problem sizes.

2.2 Hierarchical Parallelization

Parallelism exists in two phases of the random-restart IHC algorithm: between independent climbers, and between the evaluations of each 2-opt move during an IHC iteration. A handful of publications have addressed parallelizing only the local search (e.g., the 2-opt move evaluations) on the GPU. Rocki and Suda observe that at least 90% of the time during iterated local search is spent on the local search itself. They present GPU-based 3-opt [23] and 2-opt [25] local search for TSP, performing the rest of the iterated local search (which permutes local search results to generate each next starting tour rather than restarting from a random tour) on the CPU. Their implementation assigns one 2-opt swap evaluation per GPU thread. The city coordinates are copied to shared memory at the beginning of each local search phase, and atomic operations on global memory are used to determine the best 2-opt move. Similarly to the second implementation described above, distances are re-calculated as needed from the coordinates. Because the GPU performs only the local search and tours are permuted on the CPU, the city coordinates can be stored in shared memory in tour order, avoiding scattered reads and removing the need for GPU storage of a separate array of cities in tour order.

Our TSP solver is based on the key insight that the hierarchical structure of the GPU hardware suggests a hierarchical approach to parallelizing random-restart hill climbing, leveraging both parallelism between independent climbers at the block level and between the outer loop of move evaluations at the thread level. Hierarchical parallelization is in itself not new. Hybrid parallelization schemes for exploiting both task-level and loop-level parallelism on message-passing clusters of shared-memory processors are common in the literature [21]. Nested data parallelism [3] is based on the recognition that sub-computations of a data-parallel operation may themselves exhibit data parallelism, and this idea has been extended to discussions of inter- vs. intra-operator parallelism [20]. In the context of TSP, a prior GA-based TSP solver for the GPU [13] parallelized population individuals between thread blocks and the crossover operator and 2-opt local search between threads within the block. Their implementation achieves speedups of up to 24X over a serial CPU implementation of the same algorithm. Delévacq et al. present a GPU strategy for iterative local search using the 3-opt heuristic that assigns a climber per block [7]. A single thread of the block is then in charge of applying local search to the solution, but the 3-opt move evaluation is parallelized between the threads in the block. This is equivalent to parallelizing the innermost loop of the move evaluation. Their code results in a 6X speedup over serial CPU code for 2048 cities.

To our knowledge, ours is the first implementation of an iterative hill climbing TSP solver that runs entirely on the GPU and hierarchically parallelizes both the independent climbers and the outer loop of the move evaluation.

3. IMPLEMENTATION AND OPTIMIZATIONS

This section discusses our 2-opt TSP solver implementation and several of its most important code optimizations. The individual performance impact of each optimization is detailed in Section 6. Our open-source CUDA implementation is publicly available at http://cs.txstate.edu/~burtscher/research/TSP_GPU/.

Our code incorporates several optimizations from prior works. Listing 2 below illustrates the impact of these optimizations on the pseudo code for the nested move evaluation loops. Inner j-loop-invariant variables are cached in the outer i-loop of the 2-opt evaluation. At each IHC step, the code avoids calculating the full tour length by instead tracking the net change in tour length; the final tour distance is calculated only at the end of each climber's execution when a locally optimal change in length has been found. Additionally, distances between cities are recalculated from the city coordinates each time they are needed to avoid storing an $O(n^2)$ distance matrix.

```
// x[i] and y[i] are the coordinates of ith
// city in tour order
#define dist(a,b) sqrtf(
          (x[a] - x[b]) * (x[a] - x[b])
        + (y[a] - y[b]) * (y[a] - y[b]) )
do {
  minchange = 0
  for (i = 0; i < cities-2; i++) {
    minchange += dist(i,i+1)
    for (j = i+2; j < cities; j++) {
      change = dist(i,j) + dist(i+1,j+1)
               - dist(j,j+1)
      if (minchange > change) {
        minchange = change
        mini = i, minj = j
      }
    }
    minchange -= dist(i,i+1)
  }
  if (minchange < 0)
    // apply best 2-opt move
} while (minchange < 0)
```

Listing 2. Pseudo code for 2-opt evaluation loops after caching j-loop invariant variables at outer loop and re-calculating distances as needed

3.1 Intra-Parallelization of the Move Evaluation

In the random-restart IHC approach to TSP, the climbers are entirely independent, requiring only a single critical section at the end to track the global best solution. Within the climbers, the evaluation of each possible move to a neighbor (i.e., the 2-opt swaps) is also independent. However, identifying the best 2-opt swap out of the $O(n^2)$ possibilities at each IHC step requires a reduction operation, which necessitates $\log_2$(thread count) steps involving synchronization and data exchange. Our implementation exploits this hierarchical parallelism in the algorithm and maps it to the hierarchical GPU architecture. Thus, rather than assigning a climber to each thread as in previous implementations, our code assigns a climber to each thread block. Because all threads within a block share the same tour order at each IHC step and distances are recalculated as needed from the coordinates, the coordinate arrays can be permuted directly in tour order (rather than requiring a separate array of ordered city indices).

Each climber initially copies the city coordinates into local memory in parallel, accessing coordinate indices by thread index to ensure coalescing. One thread in the block is responsible for generating the initial random tour by randomly permuting the coordinate arrays. This is done serially as its runtime is insignificant. The IHC process then proceeds: each climber (block) evaluates all possible 2-opt moves, applies the best move, and repeats until a local minimum is found (i.e., until no 2-opt move will improve the tour length). At each IHC step, the threads within a block evaluate the possible 2-opt moves in parallel. Each thread is assigned one or more iterations of the outer *i*-loop. If there are fewer cities than maximum threads allowed per block (i.e., 1024), each block is launched with a thread for each city and the outer *i*-loop is fully parallelized; otherwise, the blocks are launched with 1024 threads and some or all the threads are responsible for multiple iterations of the loop. Each thread executes the iterations of the inner *j*-loop sequentially. Listing 3 illustrates the modification to the pseudo code of the nested move evaluation loops.

After all possible 2-opt moves have been evaluated, a reduction is performed between the threads in the block to determine the overall best 2-opt move from among the threads' individual best moves. This reduction requires several steps of synchronization; however, because each block executes an independent climber, this synchronization is limited to intra-block synchronization barriers. At the end of the reduction, each thread checks if its best tour modification was the best overall (i.e., if its best calculated change in tour length matches the result of the reduction). If so, the thread stores the pair of cities associated with the best 2-opt move to a buffer. Multiple threads may find 2-opt moves that yield an identical change in tour length. In case of ties, it is not necessary for the winning 2-opt move to be deterministic; however, the two city indices of the move must be stored as a single atomic transaction. In order to avoid requiring a lock, the code stores the two city indices in the upper and lower halves of a single integer value.

After another synchronization barrier to ensure that all threads have access to the best 2-opt move, the threads within the block apply the tour modification in parallel. A 2-opt swap between cities *i* and *j* results in a reversing of the segment of the tour between (and inclusive of) *i*+1 and *j*. Threads with indices within the first half of this range participate in the reordering, each swapping their assigned coordinate with the proper offset in the second half of the segment. A final synchronization barrier ends the IHC step.

When a local minimum is found by a climber, a final parallel prefix scan sums the distances between adjacent tour cities to compute the length of the locally minimal tour. The first thread in the block executes an atomicMin() to replace the global best solution if a new global minimum tour length has been found.

3.2 Shared Memory Tiling

Since each climber (block) maintains a tour that is shared by all threads within the block, the city coordinates as well as the buffer used for the parallel reductions should ideally reside in shared memory. However, storage of all city coordinates in the small

```
do {
  for (i = thdID; i < cities; i += blkDim)
    buf[i] = -dist(i,i+1)
  __syncthreads()

  minchange = 0
  // outer i-loop parallelized within block
  for (i = thdID; i < cities-2; i += blkDim){
    minchange -= buf[i]  // +dist(i,i+1)

    // inner j-loop sequential
    for (j = i+2; j < cities; j++) {
      change = buf[j] + dist(i,j)
               + dist(i+1,j+1)
      if (minchange > change) {
        minchange = change
        mini = i, minj = j
      }
    }
    minchange += buf[i]  // -dist(i,i+1)
  }
  __syncthreads()

  // reduction to identify + apply best move
} while (minchange < 0)
```

Listing 3. Pseudo code for the intra-parallelized version of the 2-opt move evaluation loops

```
#define shmem_dist(a,b) sqrtf( (shmem_x[a] - shmem_x[b]) * (shmem_x[a] - shmem_x[b])
                             + (shmem_y[a] - shmem_y[b]) * (shmem_y[a] - shmem_y[b]) )
do {
  for (i = threadID; i < cities; i += blockDim)
    buf[i] = -dist(i,i+1)
    __syncthreads()

  minchange = 0
  for (i = threadID; i < cities-2; i += blockDim) {
    minchange -= buf[i]

    // reversed j-loop order
    for (jj = cities-1; jj >= i+2; jj -= tileSize) {
      parallel_load_tile(x_shmem[], x[])
      parallel_load_tile(y_shmem[], y[])
      parallel_load_tile(buf_shmem[], buf[])
      __syncthreads()

      for(j = jj; j >= tileLowerBound; j--) {
        change = buf_shmem[j] + shmem_dist(i,j) + shmem_dist(i+1,j+1)
        if (minchange > change) {
          minchange = change
          mini = i, minj = j
        }
      }
      __syncthreads()
    }

    minchange += buf[i]
  }
  __syncthreads()
  // reduction to identify + apply best move
} while (minchange < 0)
```

Listing 4. Pseudo code for the intra-parallelized 2-opt loops with the addition of shared memory tiling

shared memory would limit the problem size. Thus, our TSP code breaks (i.e., strip mines) the inner *j*-loop move evaluations into chunks whose coordinates can fit into shared memory. The outer *i*-loop is parallelized between the threads of the block, so each thread reads its city[*i*] coordinates from global memory only once. It then executes the inner *j*-loop in segments, transferring the city[*j*] coordinates accessed by each segment into shared memory before evaluating that segment's 2-opt moves.

For shared memory tiling to work, the threads within the block must access only data falling within the same tile during the same segment of execution. The original inner *j*-loop of the 2-opt evaluations began its iteration at index *i*+2, which will vary for each thread (each of which has been assigned a different iteration of the outer *i*-loop). However, reversing the iteration order of the inner *j*-loop results in all threads beginning their loop iteration at the last city of the tour. Thus, our code executes the inner *j*-loop in reverse order, in segments beginning with the last city in the tour. Before each segment, the threads cooperate to load both the city coordinates and the loop-invariant city distances (i.e., the distance between city[*i*] and city[*i*+1] for each thread) for the segment into shared memory. After a synchronization barrier to ensure that the shared memory contains the entire tile, each thread evaluates the 2-opt moves for its assigned iteration of the *i*-loop and all *j*-loop values that fall within the tile, in decrementing order from the last city of the tour. Listing 4 illustrates in pseudo code the basic structure of the 2-opt evaluation loops with shared memory tiling. This process of tiling, synchronization, and move evaluation repeats until all segments of the inner *j*-loop have been evaluated. As individual threads reach the lower bound of their *j*-loop, which is dependent on the thread's assigned *i*-loop index, they drop out of the move evaluations. However, all threads continue participating in the parallel loading of tiles into shared memory.

After the move evaluation phase of the IHC step, the parallel reduction to determine the overall best 2-opt move is performed in shared memory. Once a climber finds a local minimum tour, a shared memory buffer is also used for the prefix sum that computes the tour length of the minimum. Whenever possible, the code reuses the same shared-memory buffer for different tasks.

3.3 Launch Configuration Tuning

Lastly, our implementation tunes the thread count per block to optimize the launch configuration based on the number of cities in the input, their shared memory usage, the maximum threads per block, the maximum threads per SM, the maximum blocks per SM, and the number of registers per SM of the target GPU.

This calculation requires only a few lines of code before the kernel launch and ensures that the hardware is maximally occupied. For many input sizes it significantly improves performance over a statically selected launch configuration (cf. Section 6).

4. EXPERIMENTAL METHODOLOGY

We evaluated our CUDA implementations on an NVIDIA K40 GPU, which has 15 SMs and a total of 2880 PEs. The codes were compiled with *nvcc* version 5.5 using the '-O3 -arch=sm_35 -use_fast_math' flags.

We measured throughput in billions of 2-opt moves evaluated per second. All codes were instrumented to collect the total number of IHC steps executed between all climbers. The number of 2-opt

evaluations per IHC step can be calculated directly from the number of cities. We measured the runtime of the entire kernel execution, exclusive of all CPU initialization code.

The throughput measurements were collected on the first *n* points of the 'd18512.tsp' input from TSPLIB [26]. For each implementation and city count, we ran a sufficient number of random independent climbers to fully load the target GPU and to result in at least one second of kernel runtime. Each measurement was performed 3 times for each input size and implementation and the best of the 3 runs was recorded. In addition to the throughput measurements, we also examined solution quality using several other TSPLIB inputs.

In addition to comparing between several CUDA implementations, we compared our 2-opt TSP solver to a parallel CPU solver based on the same IHC algorithm. Our OpenMP implementation is also hand optimized. We evaluated the OpenMP code on a single node of the TACC Maverick system, consisting of two 2.8 GHz Intel Xeon E5-2680 v2 Ivy Bridge processors with 10 cores each and 256 GB of memory. The CPU code was compiled with *icc* version 14.0.1 using the '-xhost -O3 -openmp' flags. For each input size, we ran the same number of independent CPU climbers as for the GPU implementation to which we compare the throughput, rounded up to the number of OpenMP threads.

5. INVESTIGATED CODE VERSIONS

We implemented several CUDA versions of our hierarchically parallelized TSP solver (a version with all of the optimizations described in Section 3 as well as intermediate versions to quantify the impact of individual optimizations), four CUDA codes based on the two existing GPU 2-opt TSP solvers, and an OpenMP version. This section describes each implementation evaluated in Section 6.

5.1 Distance Matrix—Shared (matr_s)

The *matr_s* code is an implementation of the O'Neil et al. [19] climber-per-thread approach that stores a distance matrix in shared memory and is limited to problem sizes of 110 or fewer cities. The distance matrix is calculated in parallel at the beginning of kernel execution. Each climber stores its tour order in a separate array of cities in local memory. This code includes all optimizations described in Section 2.1. The original O'Neil et al. code uses persistent thread blocks and requires a worklist of climbers to ensure load balance. Our re-implementation instead allows the launch of many more thread blocks than there are SMs, which enables the GPU to automatically load balance climbers via block-to-SM assignment. This simplifies the code and results in performance similar to that of the original implementation.

5.2 Distance Matrix—Global (matr_g)

The *matr_g* code is a naïve re-implementation of the O'Neil et al. algorithm that can solve larger problems. It stores the distance matrix in global memory and does not use shared memory at all. It is limited to problem sizes where the $O(n^2)$ distance matrix fits in global memory.

5.3 Distance Matrix—Global Read-Only Path (matr_g_ro)

The *matr_g_ro* code is identical to the *matr_g* code except that it accesses the distance matrix in global memory via the *__ldg()* intrinsic, which forces the read to occur through the read-only data cache (texture cache) path and allows the GPU to cache the data on the SMs.

5.4 Distance Re-Calculation (calc)

The *calc* code incorporates two of the main ideas presented in Rocki and Suda's works. It recalculates the distances between cities as they are needed rather than reading them from a precalculated distance matrix [24]. It also directly permutes the coordinate array to represent the tour order, rather than storing and reading from a separate array of cities [25]. Our implementation stores the city coordinates in local memory and does not use shared memory. Because each climber is assigned to an individual thread, permuting the city coordinates requires each thread to have its own copy of the coordinate arrays. This would not be possible in the limited amount of available shared memory. This implementation can solve problem sizes up to around 4000 cities, limited by the local memory size.

5.5 Intra-Parallelization—Global (intra)

The *intra* code is an implementation of the approach described in Section 3.1. Instead of assigning a climber to a thread, it assigns climbers to blocks and parallelizes the outer loop of the 2-opt move evaluations between threads. This implementation also does not use shared memory.

5.6 Intra-Parallelization + Shared Memory Tiling (tile)

The *tile* code parallelizes climbers between blocks and move evaluations between threads as in the *intra* code. As described in Section 3.2, it reverses the inner *j*-loop order and tiles city coordinate data into shared memory during inner loop execution.

5.7 Intra-Parallelization + Tiling + Tuned Launch Configuration (tuned)

The *tuned* code's kernel is identical to that of the *tile* code. However, this implementation adds a small section of CPU initialization code to dynamically tune the grid configuration for the given problem size and target GPU, as described in Section 3.3.

5.8 OpenMP (cpu)

The *cpu* code assigns independent climbers to OpenMP threads similarly to the *matr_s*, *matr_g(_ro)*, and *calc* codes. At each IHC step, all possible 2-opt moves are evaluated sequentially. Similarly to the CUDA code, the OpenMP code optimizes the loop nest by caching loop invariant variables in registers. The distances between cities are calculated in parallel at the beginning and stored in a distance matrix. The code stores the tour order in a separate array of city indices. A critical section at the end of a climb determines whether a new global optimum has been found.

6. RESULTS

This section presents and analyzes our experimental results. We examine throughput in billions of 2-opt moves (hereafter referred to as *gigamoves*) per second. We compare our TSP solver to the other GPU versions described above, and we also compare our best GPU version to our OpenMP implementation of the 2-opt solver. Lastly, we discuss solution quality.

6.1 GPU Codes

Figure 2 displays the throughput of the existing GPU implementations: *matr_s*, which is limited to problem sizes of 110 or fewer cities due to its $O(n^2)$ shared memory requirements; *matr_g*, which instead stores the distance matrix in global memory; *matr_g_ro*, which loads the distance matrix via the read-only data cache path; and *calc*, which re-calculates distances directly from the coordinates (stored in local memory in tour order).

At 110 cities or fewer, *matr_s* is the winner. *Matr_g* performs terribly, as expected from a naïve implementation that stores all data in global memory. Reading the global memory distance matrix via the read-only path (*matr_g_ro*), which allows the data to be cached in the read-only data cache, rivals the performance of the shared memory version at very small problem sizes. However, the performance tails off quickly as the city count grows, with a large drop in throughput at around 60 cities when the distance matrix exceeds the size of the read-only data cache and another at around 600 cities when the L2 capacity is exceeded.

At larger input sizes, *calc* achieves over 75% of the throughput of the *matr_s* code even though it does not use shared memory. Unlike the *matr_s* version, *calc* is capable of solving relatively large problem sizes (up to around 4000 cities, which is why *calc*'s throughput drops to zero at the right side of Figure 2). Clearly, smart utilization of the memory hierarchy and the GPU's computation throughput can sometimes allow good performance even in the absence of shared memory use.

Figure 3 shows the performance of our hierarchically parallelized implementation, *intra*, which parallelizes climbers between blocks and each climber's move evaluations between the threads in the block. Like *calc*, this code does not use shared memory. For small problem sizes, *matr_s* remains the best performing strategy. The *intra* code's throughput rivals that of *calc* at larger problem sizes. Unlike *calc*, which is limited to problems of up to around 4000 cities due to its need to store each thread's coordinates in tour order in local memory, the *intra* code can solve much larger problems. However, *calc* slightly outperforms *intra* for problem sizes supported by both codes due to *intra*'s multi-level parallelization strategy, which incurs a small runtime overhead.

Figure 4 illustrates the performance improvement of each of our two additional optimizations over the intra-parallelized version, once again compared to *matr_s* and *calc*. The *tile* code tiles city coordinates (in tour order) into shared memory. For the most part, at small problem sizes *matr_s* remains the winning implementation. However, at the one examined problem size over 100 cities that is supported by both codes, *tile* slightly outperforms *matr_s*. (This is because the *matr_s* code suffers from a significant number of shared memory bank conflicts, whereas the *tile* code is entirely bank conflict-free. We confirmed this behavior via the *nvprof* tool, which on the K40 counts shared bank conflicts via the *shared_load_replay* and *shared_store_replay* hardware performance counters. For example, for 100 cities and 100,000 climbers, the *matr_s* code results in about 1.3×10^{10} shared load replays compared to zero replays for the *tile* code). At larger problem sizes, *tile* significantly exceeds the maximum throughput achieved by the *matr_s* code on any input. It also significantly outperforms *calc*, achieving nearly 2.5X the throughput of *calc* on their largest shared input size.

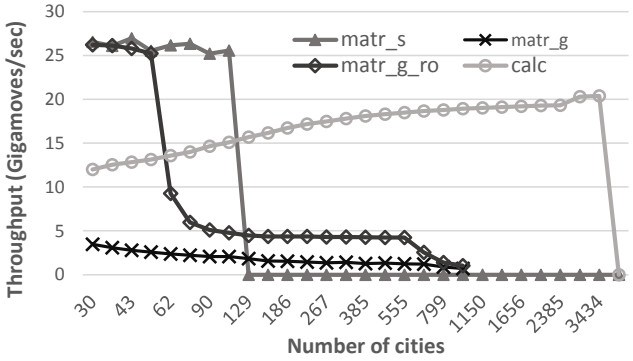

Figure 2. Throughput (in billions of 2-opt moves evaluated per second) for the existing implementations

The *tile* code suffers from some throughput variance due to suboptimal grid configurations. For example, with 555 cities, it runs 3 thread blocks per SM with 553 threads per block (one for each of the outer *i*-loop iterations), which results in poor occupancy. Our final optimized version, *tuned*, adds a small section of CPU code before the kernel launch to quickly compute the best grid launch configuration for the specific target hardware and problem size. In the 555-city case, it chooses 128-thread blocks and assigns multiple cities to each thread. *Tuned* achieves up to a 3X speedup over *calc*, reaches throughputs of over 60 billion 2-opt moves evaluated per second, and can run all problem sizes whose coordinate arrays fit in GPU memory, i.e., those with up to hundreds of millions of cities.

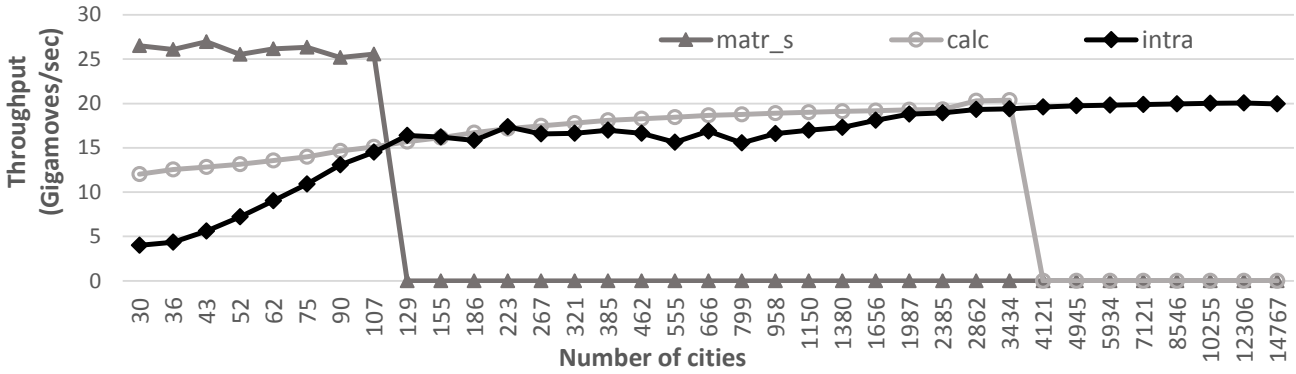

Figure 3. Throughput (in billions of 2-opt moves evaluated per second) for our intra-parallelization code

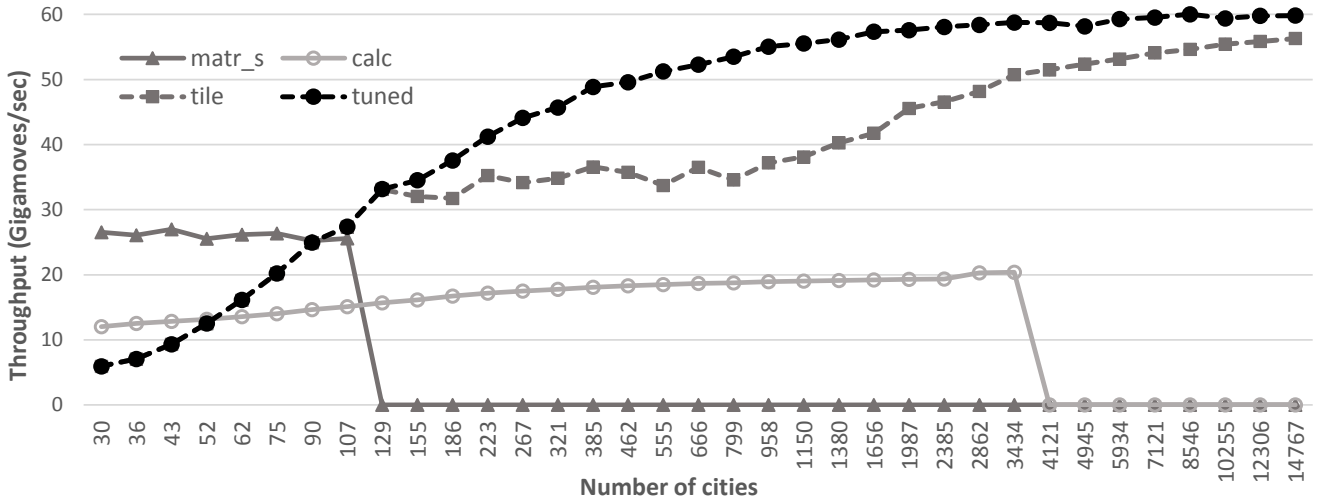

Figure 4. Throughput (in billions of 2-opt moves evaluated per second) for the shared memory tiling versions of our code

6.2 Comparison to CPU Code

We instrumented our OpenMP implementation, *cpu*, to measure its throughput in gigamoves/second identically to how it is done in the GPU code. As we evaluated our OpenMP implementation on 20 Xeon cores, we first examine performance scaling with increasing numbers of threads on the 150-city 'kroA150.tsp' input from TSPLIB. Figure 5 displays the runtime at each thread count up to the number of CPU cores, with all runtimes normalized to that of the sequential CPU code. Figure 5 also includes the normalized GPU (*tuned*) runtime on the same input.

Next we compare the performance of the CPU code (using the best thread count, i.e., 20 threads) to the GPU (*tuned*) code by comparing throughputs on increasing problem sizes. Figure 6 shows the throughput of the *tuned* and *cpu* codes on input sizes up to 8,546 cities. (Our experimental platform has a runtime limit of 12 hours. At problem sizes above 8,546 cities, the OpenMP code timed out before completion). Our *tuned* GPU code on a single K40 outperforms two Xeon E5-2680 v2 processors with twenty cores by up to 8X. The GPU implementation scales well to larger problem sizes, maintaining throughputs of over 60 billion 2-opt moves per second even at thousands of cities. The *cpu* implementation, on the other hand, achieves its maximum throughput around 1600 cities and then begins to lose throughput for larger city counts, presumably due to increased data cache misses. However, it does outperform *tuned* on the smallest tested input.

6.3 Solution Quality

Figure 7 displays the mean percentage error (i.e., the relative difference in tour length between the best found and the optimal tours) as a function of the problem size for 100,000 independent climbers. The error measurements were collected using the *tuned* code on the first 63 inputs of TSPLIB [26]. Random-restart hill climbing is one of many techniques that can be applied to TSP, and we do not claim that it is the most suitable. However, multi-start search algorithms such as IHC are applicable to broad classes of combinatorial optimization problems, and our results suggest the potential for GPUs to greatly accelerate these strategies.

Note that we have previously compared implementation performance measured as throughput in billions of 2-opt move

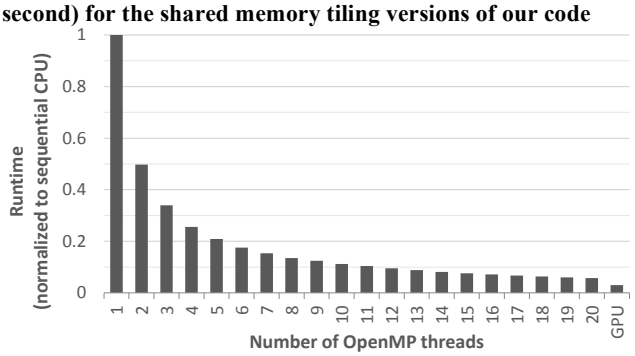

Figure 5. Runtime scaling for each tested OpenMP thread count, normalized to the runtime of the sequential CPU code

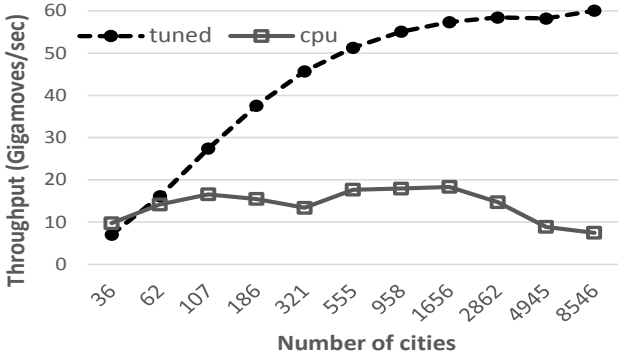

Figure 6. Throughput (in billions of 2-opt moves evaluated per second) for our best GPU implementation compared to the OpenMP code with 20 threads

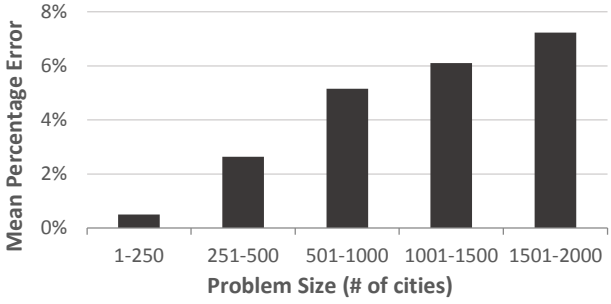

Figure 7. Solution quality (mean percentage error from optimal solution) for ranges of problem sizes

evaluations per second, but that the number of climbers launched differed between the *matr*/*calc* implementations and our proposed *tuned* implementation due to the change in the parallelization approach (from climber-per-thread to climber-per-block). Figure 8 displays the speedup of the *tuned* code relative to the *calc* code in climbers per second rather than 2-opt moves per second, collected for each implementation using the same initial tour and the same total climber count (chosen to be the minimum number of climbers required to fully load *calc*). Within the bounds of some indeterminacy for equal-length tours, both codes will yield the same solution given these matched initial conditions. The *tuned* code's much higher throughput results in a significantly higher climbers/second throughput as well, and *tuned* can therefore run more climbers in a fixed amount time. In general, solution quality improves as climber count increases.

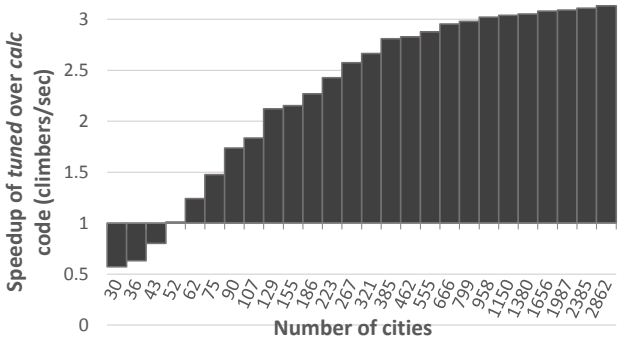

Figure 8. Speedup in climbers/second throughput of *tuned* over *calc*, measured using the same climber count and initial tour

7. CONCLUSION

This paper presents a CUDA version of a heuristic TSP solver based on random-restart hill climbing using 2-opt for local search. Our implementation includes several optimizations over existing work, including a hierarchical parallelism strategy that exploits the hierarchical nature of the GPU hardware to parallelize both independent climbers and the local search within each climb. It leverages shared memory without limiting the solvable problem size by reversing the inner loop of the move evaluation to allow for shared data tiling. It dynamically computes the best kernel launch configuration based on the input size and the underlying GPU's parameters to ensure that the hardware is maximally occupied. Our open-source CUDA implementation is publicly available at http://cs.txstate.edu/~burtscher/research/TSP_GPU/.

We present our implementation, analyze the performance impact of each of our optimizations, and compare our code's performance to that of existing approaches as well as to a parallel CPU version. Our code on a single GPU evaluates over 60 billion 2-opt moves per second, outperforming other GPU versions by 3X and an OpenMP version run on two 10-core Xeon CPUs by 8X.

Random-restart hill climbing is a common approach to address TSP and other combinatorial optimization problems. Our results suggest that a hierarchical strategy based on parallelizing both independent climbs and the local search may be the most effective technique to accelerate this algorithm on GPUs. The hierarchical climbers-to-blocks and move-evaluations-to-threads parallelization strategy described in this paper is likely also applicable to other iterated local search algorithms, e.g., those where previous local solutions are permuted to seed new climbs, which may improve solution quality.

8. ACKNOWLEDGMENTS

This work was supported by the U.S. National Science Foundation Graduate Research Fellowship Program under grant 1144466, as well as by NSF grants 1141022, 1217231, 1406304, and 1438963, a REP grant from Texas State University, and grants and gifts from NVIDIA Corporation. The authors acknowledge the Texas Advanced Computing Center for providing the HPC resources used in this study.

9. REFERENCES

[1] Agarwala, R., Applegate, D.L., Maglott, D., Schuler, G.D., Schaffer, A.A. 2000. A Fast and Scalable Radiation Hybrid Map Construction and Integration Strategy. *Genome Res.* 10, 3 (Mar. 2000), 350-364.

[2] Bai, H., OuYang, D., Li, X., He, L., and Yu, H. 2009. MAX-MIN Ant System on GPU with CUDA. *Proceedings of the Fourth International Conference on Innovative Computing, Information and Control* (Dec. 2009), 801-804.

[3] Blelloch, G. E. 1996. Programming Parallel Algorithms. *Communications of the ACM*. 39, 3 (Mar. 1996), 85–97.

[4] Cecilia, J.M., García, J.M., Nisbet, A., Amos, M., and Ujaldón, M. 2013. Enhancing Data Parallelism for Ant Colony Optimization on GPUs. *J. Parallel Distrib. Comput.* 73, 1 (Jan. 2013), 42-51.

[5] Chen, S., Davis, S., Jiang, H., and Novobilski, A. 2011. CUDA-Based Genetic Algorithm on Traveling Salesman Problem. *Computer and Information Science 2011*, R. Lee, Ed. Springer Berlin, Heidelberg, 241-252.

[6] Croes, G.A. 1958. A Method for Solving Traveling-Salesman Problems. *Oper. Res.* 6, 791-812.

[7] Delévacq, A., Delisle, P., and Krajecki, M. 2012. Parallel GPU Implementation of Iterated Local Search for the Travelling Salesman Problem. *Proceedings of the Sixth International Conference on Learning and Intelligent Optimization* (Jan. 2012), 372-377.

[8] Delévacq, A., Delisle, P., Gravel, M., and Krajecki, M. 2013. Parallel Ant Colony Optimization on Graphics Processing Units. *J. Parallel Distrib. Comput.* 73, 1 (Jan. 2013), 52-61.

[9] Dorigo, M. 1992. *Optimization, Learning and Natural Algorithms*, Ph.D. thesis, Politecnico di Milano, Italy, 1992.

[10] Dorigo, M. and Gambardella, L.M. 1997. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *IEEE Transactions on Evolutionary Computation* 1, 1 (Apr. 1997), 53-66.

[11] Exnar, F., Machač, O. 2011. The Travelling Salesman Problem and its Application in Logistic Practice. *WSEAS Transactions on Business and Economics* 8, 4 (Oct. 2011), 163–173.

[12] Felsenstein, J. 1995. PHYLIP (Phylogeny Inference Package). Distributed by the author, Department of Genetics, University of Washington, http://evolution.genetics.washington.edu/phylip.html.

[13] Fujimoto, N. and Tsutsui, S. 2010. A Highly-Parallel TSP Solver for a GPU Computing Platform. *Proceedings of the Seventh International Conference on Numerical Methods and Applications* (Aug. 2010), 264-271.

[14] Garey, M.R. and Johnson, D.S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman.

[15] Johnson, D. and McGeoch, L. 1997. The Traveling Salesman Problem: A Case Study in Local Optimization. *Local Search in Combinatorial Optimization*, E. Aarts and J. Lenstra, Eds. John Wiley and Sons, 215-310.

[16] Lenstra, J.K. and Rinnooy Kan, A.H.G. 1975. Some Simple Applications of the Travelling Salesman Problem. *Oper. Res.* 26, 4 (Nov. 1975), 717-733.

[17] Marti, R. 2003. Multi-Start Methods. *Handbook of Metaheuristics*, F. Glover and G.A. Kochenberger, Eds. Springer US, 355-368.

[18] Menger, K. 1932. Das botenproblem. *Ergebnisse eines Mathematischen Kolloquiums* 2. Teubner, Leipzig, 11-23.

[19] O'Neil, M.A., Tamir, D., and Burtscher, M. 2011. A Parallel GPU Version of the Traveling Salesman Problem. *Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications* (Jul. 2011), 348-353.

[20] Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M.A., Kaleem, R., Lee, T.-H., Lenharth, A., Manevich, R., Méndez-Lojo, M., Prountzos, D., and Sui, X. 2011. The Tao of Parallelism in Algorithms. *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Jun. 2011), 12-25.

[21] Rabenseifner, R., Hager, G., and Jost, G. 2009. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing* (Feb. 2009), 427-436.

[22] Rego, C. and Glover, F. 2002. Local Search and Metaheuristics. *Traveling Salesman Problem and its Variations*, G. Gutin and A.P. Punnen, Eds. Kluwer Academic Publishers, Dordrecht, 309-368.

[23] Rocki, K. and Suda, R. 2012. Accelerating 2-opt and 3-opt Local Search Using GPU in the Travelling Salesman Problem. *Proceedings of the 2012 International Conference on High Performance Computing and Simulation* (Jul. 2012), 489-495.

[24] Rocki, K. and Suda, R. 2012. An Efficient GPU Implementation of a Multi-Start TSP Solver for Large Problem Instances. *Proceedings of the Fourteenth Annual Conference Companion on Genetic and Evolutionary Computation* (Jul. 2012), 1441-1442.

[25] Rocki, K. and Suda, R. 2013. High Performance GPU Accelerated Local Optimization in TSP. *Proceedings of the IEEE Seventh International Parallel and Distributed Processing Symposium Workshops & PhD Forum* (May 2013), 1788-1796.

[26] TSPLIB, http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.

[27] Uchida, A., Ito, Y., and Nakano, K. 2012. An Efficient GPU Implementation of Ant Colony Optimization for the Traveling Salesman Problem. In *Proceedings of the Third International Conference on Networking and Computing* (Dec. 2012), 94-102.