A Parallel GPU Version of the Traveling Salesman Problem

Molly A. O'Neil, Dan Tamir, and Martin Burtscher
Department of Computer Science, Texas State University, San Marcos, TX

Abstract - *This paper describes and evaluates an implementation of iterative hill climbing with random restart for determining high-quality solutions to the traveling salesman problem. With 100,000 restarts, this algorithm finds the optimal solution for four out of five 100-city TSPLIB inputs and yields a tour that is only 0.07% longer than the optimum on the fifth input. The presented implementation is highly parallel and optimized for GPU-based execution. Running on a single GPU, it evaluates over 20 billion tour modifications per second. It takes 32 CPUs with 8 cores each (256 cores total) to match this performance.*

Keywords: Traveling Salesman Problem, Iterative Hill Climbing, GPGPU, Program Parallelization

1 Introduction

The traveling salesman problem (TSP) is one of the most commonly explored combinatorial optimization problems (COPs), often used as an early exploration ground for new approaches to COPs [1]. Consider a complete, undirected, weighted graph $G(V, E, W)$, where V is a set of vertices, E is a set of edges, and W is a set of edge weights. A Hamiltonian tour in G is a cycle that starts from a vertex $v_0 \in V$ and traverses all other vertices of G exactly once [1]. The symmetric TSP is a special case of the problem of finding a minimal Hamiltonian tour in a complete, undirected, planar, Euclidean, weighted graph in which the vertices represent cities, the edge weights represent the distances between the cities, and the distance from city v_A to city v_B is the same as the distance from city v_B to city v_A. The optimal TSP solution consists of the Hamiltonian tour that yields the minimum distance traveled.

Finding an optimal solution to TSP is NP-hard [2], so it is frequently approached using heuristic algorithms that find near-optimal tours. Constructive multi-start search algorithms, such as iterative hill climbing (IHC), are often applied to combinatorial optimization problems like TSP. These algorithms generate an initial solution and then attempt to improve it using heuristic techniques until a locally optimal solution, i.e., one that cannot be further improved, is reached. In each IHC step, a set of tour modifications, called *moves*, are evaluated to determine the best move [3], [4]. For instance, the tour can be adjusted by a heuristic such as 2-opt, which removes the edges (v_A,v_B) and (v_C,v_D) and adds edges (v_A,v_C) and (v_B,v_D) [1]. The IHC algorithm repeatedly chooses the best move as the next step, reducing the length of the tour until it finds a locally optimal solution, then restarts with a new initial construction. This process of local improvements and restarts continues until the solution is sufficiently good or a limit on computing resources is reached [5]. IHC is used for several problems, including finding the maximal parsimony (phylogenetics) tree (MPT), where thousands if not millions of restarts are needed to find a good solution with high probability, making this approach computationally expensive. In this paper, TSP serves as a test bed for improving IHC implementations for solving problems such as MPT.

The past decade has seen a rise in the use of graphics processing units (GPUs) as general-purpose computing devices that can efficiently accelerate many non-graphics programs, especially vector- and matrix-based codes exhibiting a lot of parallelism with low synchronization requirements. Because their hardware is primarily designed to perform complex computations on blocks of pixels at high speed and with wide parallelism, GPU architectures differ substantially from conventional CPU hardware. This can make it difficult to write efficient implementations of non-graphics algorithms for GPUs.

For example, NVIDIA GPUs require sets of 32 program threads, called warps, to execute the same instruction in every clock cycle or wait. When not all threads in a warp can execute the same instruction, the warp is subdivided by the hardware into sets of threads such that all threads in a set execute the same instruction. These sets execute serially until they re-converge, resulting in a loss of parallelism.

The memory subsystem is also optimized for warp-based processing. If a warp accesses 32 consecutive words in memory, the hardware merges the 32 reads or writes into one coalesced memory access that is as fast as a single non-coalesced access, subject to alignment and word-size constraints. Thus, it is crucial to use coalesced memory accesses to exploit the GPU's high memory bandwidth.

The 32 processing elements (PEs) within each streaming multiprocessor (SM) of a GPU share a pool of threads called a thread block, synchronization hardware, and a software-controlled cache called shared memory. A warp can simultaneously access up to 32 distinct words in shared memory as long as the words reside in different memory banks. Barrier synchronization between the threads in an SM takes one clock cycle if all threads reach the barrier together.

The PEs are fed with warps for execution in multi-threading style to hide latencies. Thus, it is paramount for

good performance to have many active resident warps in each SM. In other words, GPUs require thousands of simultaneously running threads, i.e., large amounts of parallelism to achieve maximum performance.

The SMs operate largely independently. They can only communicate with each other through global memory (DRAM). Thus, synchronization between SMs must be done using atomic operations on global memory locations, meaning that GPUs are most effective at accelerating codes with low sharing requirements.

The large amount of parallelism and wide memory buses make GPUs well suited to speed up codes displaying high computational intensity and little synchronization. For such codes, GPUs have demonstrated a substantial advantage over CPUs in terms of performance per dollar and performance per transistor [6] as well as performance per watt [7]. GPU implementations of these applications can be dozens of times faster than optimized parallel CPU implementations [8].

This paper explains how we parallelized and optimized the IHC algorithm for TSP so that it can reap the benefits of GPU acceleration. Our implementation running on one GPU chip is 62 times faster than the corresponding serial CPU code, 7.8 times faster than an 8-core Xeon CPU chip, and about as fast as 256 CPU cores (32 CPU chips) running an equally optimized pthreads implementation. For symmetric, planar, 100-city problems with 100,000 random restarts, our code finds the optimal solution for four out of five TSPLIB inputs and is 0.07% off on the fifth input. Our open-source CUDA implementation is freely available for download at http://www.cs.txstate.edu/~burtscher/research/TSP_GPU/.

2 Parallelization and optimization

This section explains how we implemented, optimized, and parallelized the IHC algorithm for the TSP problem. In this discussion, we assume symmetric 100-city problems with 100,000 random restarts.

2.1 Parallelization

There are several ways to parallelize this algorithm. The 100,000 climbers are independent and can be processed in any order, including concurrently. However, load balance is a potential problem when parallelizing the climbers as they require different numbers of IHC steps to reach a local optimum. Within a climber, each IHC step depends on the previous step and therefore has to execute serially. In our implementation, every IHC step evaluates 4851 opt-2 moves. These moves are independent and can be run in parallel, but they require a reduction operation at the end to determine the move that yields the largest reduction in tour length. This reduction can be performed in $\log_2(4851) \approx 13$ steps but necessitates synchronization and data exchange, which may be slow.

Because modern GPUs require tens of thousands of parallel threads that perform very similar tasks to unleash their full performance, we decided to run the independent climbers in parallel. This approach results not only in the highest degree of parallelism but also in the least amount of synchronization and data exchange. However, the climbers perform varying numbers of IHC steps to reach a local optimum. We measured between 84 and 124 steps with an average of 103.3. Since we launch 14,336 threads on the GPU, the average thread processes only 7 climbers, which results in load imbalance and consequently poor scaling. In contrast, we launch no more than 256 threads on the CPU, yielding an average of 391 climbers per thread, which is enough to average out the number of IHC steps performed by each thread. Thus, load balance is not an issue with the CPU code but is significant in the GPU code. Because load balancing imposes synchronization and serialization overheads, the pthreads code actually runs faster without load balancing whereas the CUDA code runs faster with load balancing. Hence, we ended up with the following implementations.

Our pthreads code statically assigns equal (±1) numbers of climbers to each thread. The threads run independently to find the best solution among their climbers and only execute a single critical section at the end to determine the best solution among all threads. The GPU code, in contrast, only assigns a single initial climber to each thread. When a local minimum is reached, the thread checks whether this minimum is smaller than the currently best solution. If it is, the best solution is updated using an atomic compare-and-swap instruction. Then, the next climber is obtained from a global worklist using an atomic increment. Threads terminate when the worklist is empty.

2.2 Code optimization

Our serial, pthreads, and CUDA implementations use essentially identical code for evaluating the opt-2 moves, which takes the vast majority of the runtime. This code section comprises two nested **`for`** loops that iterate over the cities to form pairs of cities between which the tour is reversed. The CUDA code differs from the serial and pthreads code in that we manually moved two loop-invariant computations out of the inner loop and specified that the inner loop be unrolled eight times. This was not necessary in the serial and pthreads codes as the C compiler automatically performs these optimizations.

Because GPUs are only fast if sets of 32 threads, i.e., warps, perform the same work (on different data) at the same time, our implementation always considers 4851 city pairs in each IHC step. In particular, the outer i-loop iterates from the 1st to the 98th city while the inner j-loop iterates from the i+2nd to the 100th city. Note that this approach avoids duplications in city pairs due to symmetry as well as pairs of adjacent cities that never result in a change of the tour length. Note that we always compute the tour length for all 4851 city pairs, including the ones that did not change from

the previous IHC step, because re-computing them is faster than recording and retrieving this information.

We optimized the loop nest by saving values fetched from memory in register variables so that later iterations can quickly access them. For example, even though we need four city IDs (the i^{th}, $i+1^{st}$, j^{th}, and $j+1^{st}$) in every iteration, the inner loop body only fetches the $j+1^{st}$ city ID from memory as the remaining values have been fetched earlier and are "cached" in variables. Similarly, each opt-2 move needs four distance values from a two-dimensional matrix (i^{th} to $i+1^{st}$ city, j^{th} to $j+1^{st}$ city, i^{th} to j^{th} city, and $i+1^{st}$ to $j+1^{st}$ city, where the 101^{st} city is the same as the 1^{st} city). Nevertheless, the code only fully evaluates one distance, partially evaluates two of the distances (by accessing a vector, i.e., a predetermined row of the matrix), and uses a cached value for the fourth distance to minimize computations and memory accesses. Aside from these operations, the inner loop only contains assignment statements that copy one scalar variable into another and an `if` statement to check whether a new optimum has been found.

To further boost the performance, the loop nest never actually computes the tour cost. It only calculates how much shorter an opt-2 move makes the current tour and picks the move that results in the greatest savings. As long as an IHC step results in a reduction in tour cost, the corresponding best opt-2 move is applied, i.e., the selected tour segment is reversed, and the next IHC step is initiated. Only once a local optimum has been reached is the tour cost finally computed. If this cost is lower than the previously found shortest tour, the new tour is written back to global memory and the shortest tour is updated. Otherwise, the new tour is simply discarded to avoid unnecessary memory writes.

To make the results deterministic and to simplify verification, the random seed used for generating a tour is the tour number (0 to 99,999). This guarantees that the length of the shortest tour is always the same, no matter in which order the 100,000 tours are processed. Because a cyclic rotation of a tour does not yield a new tour, the first city, which is also the last city, can be fixed without loss of generality. This enables simplifying and accelerating the program by hard coding the ID of the first city. Our code contains several other minute enhancements.

The CUDA code further contains GPU-specific optimizations that do not apply to the CPU code. For instance, the two-dimensional distance matrix is allocated in shared memory, a software-controlled cache, so that accesses to it are always fast. The 1024 tours that are evaluated concurrently in an SM are too large to fit in shared memory. Thus, we allocate them in global memory (DRAM). To still be able to access them quickly, the code first copies the tours into local memory, which is part of the global memory but ensures that every tour access in the two nested loops is fully coalesced. Other GPU optimizations include limiting thread divergence to rarely executed code sections and minimizing CPU/GPU transfers to just 40 kB initially to copy the distance matrix to the GPU and 108 bytes in the end to copy the best tour, its cost, and its tour number back to the CPU. Note that, other than generating the distance matrix and printing the result, our implementation runs the entire TSP algorithm on the GPU.

3 Related work

Most previous GPU-based approaches to the traveling salesman problem use the Max-Min Ant System (MMAS) algorithm [9]. This algorithm is a variant of Ant Colony Optimization (ACO), a metaheuristic algorithm based on the natural ability of ants to discover, collaboratively, the shortest path between their nest and a food source by depositing pheromone along their traveled paths. ACO algorithms simulate the behavior of individual ants, which construct tours around a graph based on the strength of evaporating pheromone trails left by other ants. Dorigo and Gambardella first presented this algorithm applied as a distributed TSP solver [10]. ACO algorithms spend the majority of their computation time in the tour construction phase [11], and because ants travel independently and each ant constructs a complete solution based only on the previous iterations' pheromone matrix, this phase is highly parallelizable.

Bai et al. detail a CUDA implementation of the parallel MMAS algorithm in which multiple ant colonies are simulated concurrently on the GPU, one for each thread block, with the tours of individual ants within each colony also parallelized [12]. This implementation achieves up to a 32x speedup over a serial CPU version under the same workload, though without finding the optimal solution in some cases. Jiening et al. present a C++ and Cg implementation of the MMAS algorithm with up to a 1.4x speedup over the CPU implementation, which finds the optimal tour on a 30-city input [13]. You describes a CUDA implementation of a parallel ACO algorithm [14], with each thread on the GPU responsible for the travel of a single ant from a unique starting location, achieving up to a 20x speedup over a serial CPU implementation. Cecilia et al. present several GPU-based, data-parallel strategies for both the tour construction and pheromone update stages of the ACO algorithm, achieving a 28x speedup for the tour stage and a 20x speedup for the pheromone update stage over sequential CPU code [15]. Many of the prior works on GPU-based ant colony solvers compare solution quality only against a serial ACO implementation and do not address how often either implementation discovers the optimum TSP solution.

There are also heterogeneous implementations of ACO algorithms, which implement only part of the TSP solver on the GPU. Delévacq et al. implement a parallel approach to the MMAS algorithm that performs tour construction on the GPU and pheromone update on the CPU [11]. Next, they compare their implementation against their GPU version of the original ACO algorithm [16], achieving better solution quality (though still suboptimal in some cases) and up to a 3.6x speedup. Fu et al. describe an MMAS implementation in MATLAB with the tour construction performed on the

GPU and the updates performed on the CPU [17]. This implementation achieves roughly a 32x speedup over sequential CPU code, but with slightly lower solution quality compared to the CPU implementation.

There also exists a recent genetic algorithm-based TSP solver in CUDA, presented by Fujimoto and Tsutsui in 2011 [18]. This work parallelizes TSP using the genetic crossover operator and 2-opt local search. Their CUDA implementation on a GTX-285 is up to 24.2x faster than a single-core CPU version, allowing an error ratio over the optimal trip cost of up to 0.5%.

To the best of our knowledge, this paper presents the first GPU implementation of the IHC algorithm for solving the TSP problem. Our IHC approach may be better suited for GPUs than previously proposed algorithms as it yields larger speedups over both serial and parallel CPU implementations while, at the same time, achieving very high solution quality.

4 Evaluation methodology

We evaluated our GPU implementation of TSP on an NVIDIA Tesla C2050 graphics card, which has CUDA compute capability 2.0 [19]. This GPU is equipped with 14 streaming multiprocessors (SMs), each with 32 cores, for a total of 448 cores running at 1.15 GHz and sharing 3 GB of global memory. Each multiprocessor is configured with 48 kB of shared memory and a 16 kB L1 cache. All SMs share a 768 kB L2 cache. Each SM has 32,768 registers that are shared among the threads allocated to the multiprocessor. The CUDA code was compiled with *nvcc* version 3.2 using the "-O3 -arch=sm_20" flags.

We ran the pthreads and sequential CPU implementations on the Nautilus supercomputer at NICS, which contains 128 2.0 GHz 8-core Xeon X7550 CPUs sharing 4 TB of main memory. The pthreads and sequential codes were compiled with *icc* version 11.1, with the "-O3 -xW -pthread" flags for the pthreads version and the "-O3 -xW" flags for the sequential version.

We instrumented the three implementations to measure the runtime of everything except the reading in of the 100 city coordinates and the generation of the distance matrix from these coordinates. We tested all implementations on five TSPLIB benchmarks containing 100 cities [20].

5 Results

Figure 1 plots the runtimes (in milliseconds) of our three IHC implementations on the kroE100 TSPLIB input, with the minimum, median, and maximum runtime of three runs plotted separately. The runtimes for other 100-city inputs and different random restarts are very similar. The median runtime is listed above the columns. The results show that our GPU implementation's median runtime, at 2.497 seconds, is slightly under that of the parallel CPU version run with 256 threads and dramatically less than that produced by the sequential CPU code (2.58 minutes). Unlike the GPU version, which produces highly consistent runtimes, the pthreads runtime at higher thread counts varies substantially between executions. In fact, in some experiments, it already started varying with 16 threads, i.e., the problem seems to appear as soon as multiple CPU chips are used. Since we made sure that there is no false sharing and only a minimal amount of true sharing in our pthreads implementation, we assume the variability is caused by interference from other jobs that were running at the same time on this large shared memory machine.

Figure 2 displays the minimum, median, and maximum speedups of the pthreads and GPU implementations relative to the sequential CPU implementation. Again, we see that

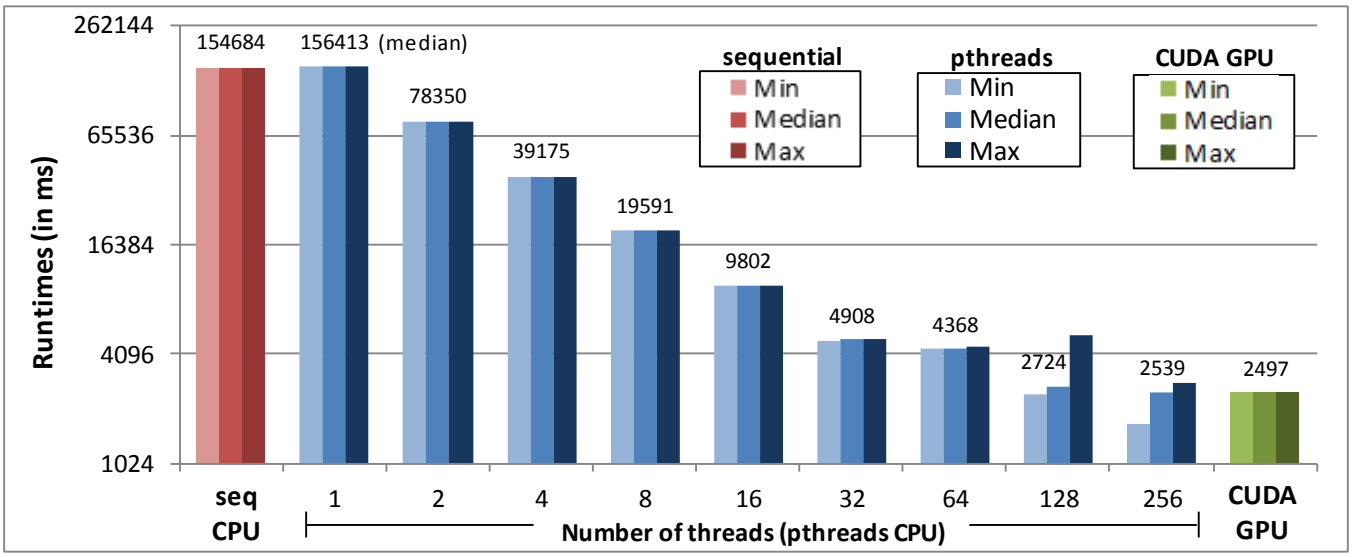

Figure 1. Minimum, median, and maximum runtimes (in milliseconds) of the three TSP implementations (note that this graph is logarithmic)

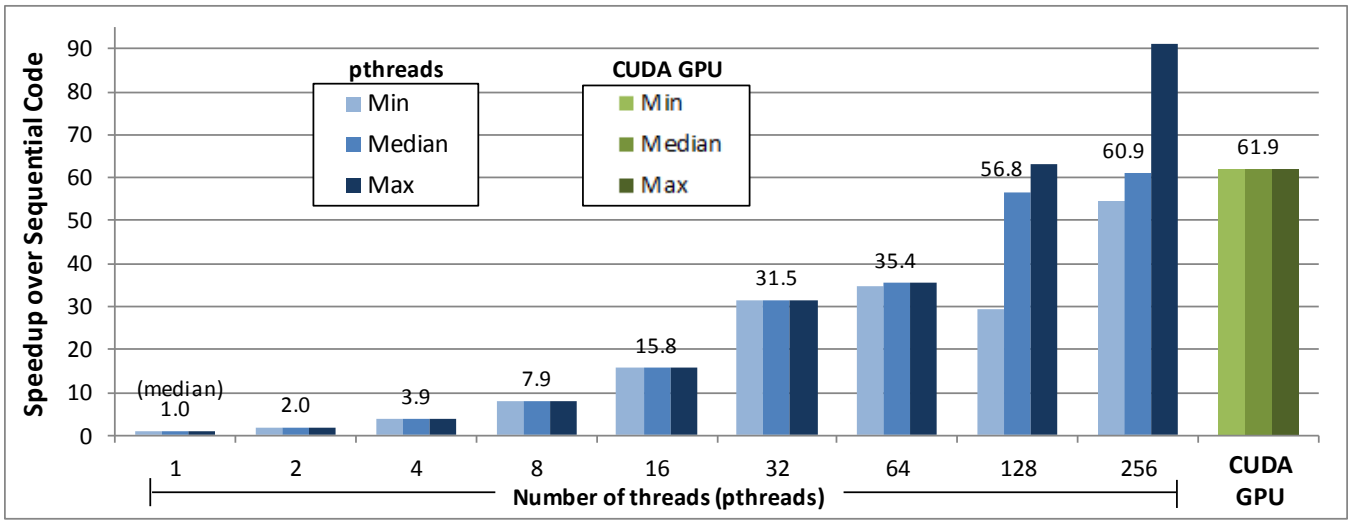

Figure 2. Minimum, median, and maximum speedup of the pthreads and GPU implementations relative to the serial CPU implementation

the GPU version produces consistent speedups whereas the pthreads version with 128 and 256 threads demonstrates significant performance variance. The pthreads code scales almost perfectly to 32 cores, indicating that it does not suffer from false sharing, load imbalance, serialization, or other parallelization overheads. However, scaling is poor beyond 32 threads, possibly due to the increasing thread startup cost. Additional experiments with different random restarts resulted in the same scaling trends. While the maximum speedup offered by the 256-thread CPU version exceeds that of the GPU implementation, the GPU code outperforms the pthreads code in terms of median speedup. It achieves a consistent speedup of around 61.9 compared to the 256-thread pthreads version's median speedup of 60.9. This means that the GPU is capable of slightly exceeding the performance of 256 x86 cores or 32 CPUs with eight cores each on the IHC TSP algorithm.

The Nautilus supercomputer on which we tested the pthreads implementation has 2.0 GHz CPU cores. The sequential and pthreads implementations would benefit from CPUs with faster clocks. However, we found the GPU implementation to still offer a 50x speedup over the sequential implementation executed on a 2.53 GHz Intel Xeon, suggesting that the GPU solution offers a large performance advantage over the CPU implementation even for the fastest currently available CPUs.

Table 1 addresses the solution quality and shows the cost and number of the shortest tour found by the GPU implementation for five 100-city inputs from the TSPLIB library when using 100,000 random restarts. The optimal tour cost and the runtime for each input are shown as well. Our GPU code finds the optimal tour in all but one case, on kroE100, where the tour is 0.07% longer. Doubling the number of climbers to 200,000 allows the GPU code to find the optimal tour in the last case as well.

Table 1. Solution quality achieved by the GPU implementation for five 100-city inputs from TSPLIB

TSPLIB Database		CUDA GPU Solution Quality		
Name	Optimal Cost	Min. Tour Cost	Min. Tour #	Runtime (s)
kroA100	21,282	21,282	33,188	2.540
kroB100	22,141	22,141	5,969	2.499
kroC100	20,749	20,749	23,092	2.543
kroD100	21,294	21,294	32,142	2.497
kroE100	22,068	22,084	16,941	2.499
		22,068	*117,583*	*4.952*

6 Summary and conclusions

This paper explains how we parallelized and optimized the IHC algorithm for solving the TSP problem on GPUs. The results demonstrate that our implementation not only yields a high solution quality but also runs very quickly. It processes over 20 billion 2-opt moves per second on a single GPU, which is 62 times faster than an x86 core and as fast as 32 CPUs with 8 cores running a pthreads version of the same algorithm. Based on these results, we believe our approach may be better suited for GPU-based acceleration than the related ant colony and genetic algorithm-based TSP solvers that are available for GPUs.

7 Acknowledgments

This research was supported by an allocation of advanced computing resources provided by the National Science Foundation. Some of the computations were performed on Nautilus at the National Institute for Computational Sciences [21]. We thank NVIDIA Corporation for donating the GPU that was used to develop,

tune, and measure the CUDA implementation of the algorithm presented in this paper. We further thank Intel Corporation for donating the server on which the serial and pthreads codes were developed.

8 References

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

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

[3] Ambite, J. and Knoblock, C. "Planning by Rewriting." *Journal of Artificial Intelligence Research*, pp. 207-261. 2001.

[4] Pitsoulis, L.S. and Resende, M.G.C. "Greedy Randomized Adaptive Search Procedures." *Handbook of Applied Optimization*. Oxford University Press, pp. 168-183. 2001.

[5] Rego, C. and Glover, F. "Local Search and Metaheuristics." *The Traveling Salesman Problem and its Variations*, by G. Gutin and A.P. Punnen (Eds.), pp. 309-368. Dordrecht: Kluwer Academic Publishers, 2002.

[6] Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., and Purcell, T.J., "A Survey of General-Purpose Computation on Graphics Hardware." *Computer Graphics Forum*, Vol. 26, pp. 80-113. 2007.

[7] Huang, S., Xiao, S., and Feng, W. "On the Energy Efficiency of Graphics Processing Units for Scientific Computing." *International Symposium on Parallel Distributed Processing*, pp. 1-8. 2009.

[8] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., and Skadron, K. "A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA," *Journal of Parallel and Distributed Computing*, Vol. 68, No. 10, pp. 1370-1380. 2008.

[9] Stutzle, T. and Hoos, H.H. "MAX-MIN Ant System." *Future Gen. Comput. Syst.*, vol. 16, no. 9, pp. 889-914. June 2000.

[10] Dorigo, M. and Gambardella, L.M. "Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem." *IEEE Transactions on Evolutionary Computation*, Vol. 1, No. 1, pp. 53-66. April 1997.

[11] Delévacq, A., Delisle, P., and Krajecki, M. "Max-min Ant System on Graphics Processing Units." *Third International Conference on Metaheuristics and Nature Inspired Computing*. October 2010.

[12] Bai, H., Yang, D.O., Li, X., He, L., and Yu, H. "MAX-MIN Ant System on GPU with CUDA." *Fourth International Conference on Innovative Computing, Information and Control*, pp. 801-804. December 2009.

[13] Jiening, W., Jiankang, D., and Chunfeng, Z. "Implementation of Ant Colony Algorithm Based on GPU." *Sixth International Conference on Computer Graphics, Imaging and Visualization*, pp. 50-53. August 2009.

[14] You, Y.-S. "Parallel Ant System for Traveling Salesman Problem on GPUs." *Eleventh Annual Conference on Genetic and Evolutionary Computation*. July 2009.

[15] Cecilia, J.M., Garcia, J.M., Ujaldon, M., Nisbet, A., and Amos, M. "Parallelization Strategies for Ant Colony Optimisation on GPUs." *14th International Workshop on Nature Inspired Distributed Computing*. May 2011.

[16] Delévacq, A., Delisle, P., Gravel, M., and Krajecki, M. "Parallel Ant Colony Optimization on Graphics Processing Units." *Sixteenth International Conference on Parallel and Distributed Processing Techniques and Applications*. July 2010.

[17] Fu, J., Lei, L., and Zhou, G. "A Parallel Ant Colony Optimization Algorithm with GPU-Acceleration Based on All-in-Roulette Selection." *Third International Workshop on Advanced Computational Intelligence*, pp. 260-264. August 2010.

[18] Fujimoto, N. and Tsutsui, S. "A Highly-Parallel TSP Solver for a GPU Computing Platform." *Lecture Notes in Computer Science*, Vol. 6046, pp. 264-271. 2011.

[19] "NVIDIA's Next Generation CUDA Compute Architecture: Fermi." Whitepaper, NVIDIA Corporation. 2009.

[20] Reinelt, G. "TSPLIB—A Traveling Salesman Problem Library." *ORSA Journal on Computing*, Vol. 3, No. 4, pp. 376-384. Fall 1991.

[21] National Institute for Computational Sciences, http://www.nics.tennessee.edu/. Last accessed March 8, 2011.