An Efficient GPU Implementation of a Multi-Start TSP Solver for Large Problem Instances

Kamil Rocki
Department of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo, CREST, JST
7-3-1, Hongo, Bunkyo-ku
113-8656 Tokyo
kamil.rocki@is.s.u-tokyo.ac.jp

Reiji Suda
Department of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo, CREST, JST
7-3-1, Hongo, Bunkyo-ku
113-8656 Tokyo
reiji@is.s.u-tokyo.ac.jp

ABSTRACT

We are presenting a parallel GPU implementation of the Traveling Salesman Problem (TSP) solver. Our method is based on the iterative hill climbing algorithm first proposed by O'Neil et al., but modified in order to solve large instances of the problem. Our results show that GPU can be used to find an approximate solution of a problem which size is up to 6 thousand cities in a very efficient way achieving over 170 GFLOPS on a single TESLA C2050 card running thousands of independent threads. Due to computation-oriented architecture of GPUs, a more universal algorithm can be obtained with relatively little performance degradation.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

Keywords

Traveling salesman problem, GPU, Iterative Local Search

1. INTRODUCTION

The traveling salesman problem (TSP)[1][2] is one of the most widely studied combinatorial optimization problem and therefore it has become a testbed for new algorithms as it is easy to compare results among the published works. It is also very simple - for a given number of cities N, find the shortest path that visits all N cities exactly once. Recently the use of graphics processing units (GPUs) as general purpose computing devices has risen significantly, accelerating many non-graphics programs, exhibiting a lot of parallelism with low synchronization requirements. Moreover, the current trend in modern supercomputer architectures is to use GPUs as low-power consuming devices. Therefore, for us it is important to analyze possible application of CUDA (Compute Unified Device Architecture [7]) as a GPU programming language to speedup optimization and provide a basis for future similar applications using massively parallel systems. In our work we analyzed several GPU implementations and optimizations which make the code significantly faster than the basic naive approach. 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. O'Neil et al. [3] describe and evaluate a parallel implementation of iterative hill climbing with random restart for determining high-quality solutions to the traveling salesman problem. Their implementation running on one GPU chip was 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. Our research was inspired by the results presented in that paper, yet our approach to parallelize the search is slightly different and the results are better in our opinion. Their approach, although fast, is limited by the GPU's shared memory size and the largest problem's size that can be tackled is 110 cities. Most of the other works related to parallel GPU TSP solvers regards evolutionary and genetic programming, such as Genetic Algorithms (GA) [4]. The original GPU implementation (O'Neil et al.)[3] using IHC and multiple *climbers* performs quite well. Their algorithm pre-calculates the distances between the cities on the CPU side and stores them in the *shared memory* on GPU. Access to the *shared memory* is very fast, therefore data stored in *shared memory* can be accessed with very low latency. However, due to the GPU limitations (48kB of *shared memory [[7] - Appendix F]* per *MultiProcessor* to store pre-calculated distances between the cities) that algorithm is unable to solve instances larger than 110 cities. In their approach, the code is optimized for problems which size is not larger than 110 cities. Many of the parameters are fixed according to the that assumption. In our opinion, this causes the implementation to be be highly impractical. Nowadays, a more universal and generalized algorithm is needed and 100-city instances can be solved fairly quickly. Our goal was to have an implementation that could be applied to much more complex problems and parallelized even more using multiple GPUs. We propose a solution for this issue. Instead of pre-calculating the distances and taking advantage of fast shared memory, high computational power of modern GPUs can be exploited by calculating the distances 'on-the-fly'. Shared memory in this case can be used to store the coordinates, which also can be

GECCO'12 Companion, July 7–11, 2012, Philadelphia, PA, USA.
ACM 978-1-4503-1178-6/12/07.

a limitation, but up to 6000-city instances can be solved. Our approach removes the 110-city limitation, but the code is approximately 2 times slower compared to the original algorithm.

2. RESULTS AND ANALYSIS

We performed runs on predefined problems from TSP library (TSPLIB [Reinelt 1991]) and evaluated our GPU implementation of TSP on NVIDIA TESLA C2050 GPU with CUDA 4.0 installed. Table 1 present a comparison between the original algorithm by O'Neil et al.[3] and ours. The first and the most natural implementation in order to remove the 110-city limitation was not to use the shared memory storing pre-calculated distances. Because of the latency of the global memory, we tried to apply some modifications in the code to take advantage of other GPU properties. We decided not to calculate the data by CPU, but just to read the coordinates and transfer them to the GPU to the fast on-chip shared memory for low-latency access. In this case, the storage limit is approximately 6,144 cities (48 kB/ 8B [2x float type]). Using less precise *sqrt* function and *Fast math* compilation option led to usage of faster equivalents of mathematical functions on the GPU and shortened the time. Finally we achieved 0.746s search time, which was still 90% longer when compared to the original algorithm, but the problem size limit has been increased from 110 to over 6000. The next table (Table 2) shows results of runs using different algorithms, problem sizes and number of starting points. We introduced 2 types of new performance indicators to measure the differences. The first one is FLOPS (Floating OPeration per Second). And the second one is the number of 2-opt exchanges per second (EX/s). It can be observed that the GPU is very fast in terms of FLOPS, and that justifies its usage to calculate the distances rather than just reading them from the memory. As the problem size increased, we saw that the performance was slightly better as well (from 109.4 GFLOPS to 117.1 GFLOPS in case of 10000 *climbers* and from 167.7 GFLOPS to 172.8 GFLOPS in case of 100000 *climbers*).

Table 1: Results - time needed to reach an approximate solution - kroE100.tsp, 10000 starting points, 14 blocks, 1024 threads per block

Method	time
Precalculated distances (shared memory - original)	0.3955s
Precalculated distances (global memory)	6.0079s
Calculated (coordinates in the shared memory)	1.2003s
Calculated (coordinates in the shared memory) (less precise sqrt + fast math)	0.7451s

3. CONLUSIONS AND FUTURE WORK

Our results show that GPU can be used to find an approximate solution of a problem which size is up to 6 thousand cities in a very efficient way achieving over 170 GFLOP per second on a single TESLA C2050 card running thousands of independent threads. The main conclusion is that due to computation-oriented architecture of GPUs, a more universal algorithm can be obtained with relatively little performance degradation. We showed that our algorithm performs well on large TSP instances (tested up to 1084-city problem). Currently, we are planning to obtain a scalable implementation to be able to get significant speedup for problems larger than 1000 cities on the TSUBAME 2.0 supercomputer using hundreds of available GPUs. In order to do this we are working on parallelizing the 2-opt search itself on GPU.

Table 2: Time (s) / FLOPS / 2opt EXchanges/s

TSPLIB Instance (starting points)	Pre-calculated shared memory (original algorithm)	Runtime calculation (coordinates in shared memory)
kroE100.tsp (100000) 0.07% err	**2.58s** 2.95M EX/s	5.2s 1.47M EX/s 167.7 GFLOPS
kroA200.tsp (100000) 1.28% err	Limited to 110 cities	**45.6s** 370.2K EX/s 170.5 GFLOPS
rat783.tsp (100000) 7.7% err	Limited to 110 cities	**2928.4s** 24.9K EX/s 172.7 GFLOPS

4. ACKNOWLEDGMENTS

Our work was based on partial code provided by M. A. O'Neil et al.[3][1]. We refer to that code as *'The Original Implementation'*.

5. REFERENCES

[1] Applegate, D.L., Bixby, R.E., Chvatal, V., Cook, W.J.: The Traveling Salesman Problem: A Computational Study. Princeton University Press, Princeton (2007)

[2] Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., Shmoys, D.B.: The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. Wiley, Chichester (1985)

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

[4] 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.

[5] Reinelt, G.: TSPLIB - A Traveling Salesman Problem Library. ORSA Journal on Computing, Vol. 3, No. 4, pp. 376–384. Fall 1991.

[6] Lourenco, H. R. Martin, O. C. Stutzle, T.: Iterated Local Search, International series in operations research and management science, 2003, ISSU 57, pages 321–354

[7] NVIDIA CUDA Programming Guide, http://developer.download.nvidia.com

[1] `http://www.cs.txstate.edu/~burtscher/research/TSP_GPU/`