Accelerating 2-opt and 3-opt Local Search Using GPU in the Travelling Salesman Problem

Conference Paper · July 2012

DOI: 10.1109/CCGrid.2012.133

2 authors:

Kamil Rocki
The University of Tokyo
12 PUBLICATIONS **97** CITATIONS
SEE PROFILE

Reiji Suda
The University of Tokyo
95 PUBLICATIONS **604** CITATIONS
SEE PROFILE

Accelerating 2-opt and 3-opt Local Search Using GPU in the Travelling Salesman Problem

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

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

***Abstract*—In this paper we are presenting high performance GPU implementations of the 2-opt and 3-opt local search algorithms used to solve the Traveling Salesman Problem. The main idea behind them is to take a route that crosses over itself and reorder it so that it does not. It is a very important local search technique. GPU usage greatly decreases the time needed to find the best edges to be swapped in a route. Our results show that at least 90% of the time during Iterated Local Search is spent on the local search itself. We used 13 TSPLIB problem instances with sizes ranging from 100 to 4461 cities for testing. Our results show that by using our GPU algorithm, the time needed to find optimal swaps can be decreased approximately 3 to 26 times compared to parallel CPU code using 32 cores. Additionally, we are pointing out the memory bandwidth limitation problem in current parallel architectures. We are showing that re-computing data is usually faster than reading it from memory in case of multi-core systems and we are proposing this approach as a solution.**

I. Introduction

A. *Traveling salesman problem*

The traveling salesman problem (TSP)[3][4][2] is one of the most widely studied combinatorial optimization problems. It has become a testbed for new algorithms as it is easy to compare the results among the published works. This problem is classified as NP-hard[5], with a pure brute force algorithm requiring factorial time. In the symmetric TSP, the distance between two cities is the same in each opposite direction, forming an undirected graph. This symmetry halves the number of possible solutions. The most direct solution for a TSP problem would be to calculate the number of different tours through *n* cities. Given a starting city, it has *n-1* choices for the second city, *n-2* choices for the third city, etc. In the asymmetric TSP, paths may not exist in both directions or the distances might be different, forming a directed graph. A large number of heuristics have been developed that give reasonably good approximations to the optimal tour in polynomial time. One of the most well known methods of solving the problem (approximately) is by repeating a series of steps called 2-opt exchanges [1]. Iterated Local Search (ILS) algorithm, as it is called, uses local search and a random perturbation after finding a local minimum within certain neighborhood, so that it avoid being stuck in local minima and finding the global solution is guaranteed, given search time is infinite. Our results show that at least 90% of the time during Iterated Local Search is spent on the 2-opt itself (Figure 1) and that number increases with the problem size growing. This is easy to

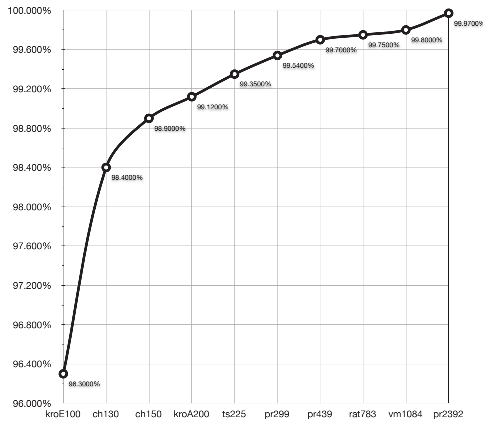

Fig. 1: Percentage of time spent on 2-opt local search only during a single Iterative Local Search run

explain as the complexity of the 2-opt search is $O(n^2)$ and any simple perturbation method would be of order $O(n)$, where n is the number of the cities. Therefore, in order to solve the whole problem quickly using the iterative approach, this basic step has to be optimized. An important thing here that should be mentioned is that, we focus only on the 2-opt and 3-opt single functions, not on the whole search algorithm, where these 2 or 3-

opt exchanges are repeated many times. However, we are certain, that by speeding up these single steps, the run time of whole algorithm will be definitely shorter (that is a subject of our current ongoing research). Recently the use of graphics processing units (GPUs) as general-purpose computing devices has risen significantly, accelerating many non-graphics programs that exhibit a lot of parallelism with low synchronization requirements. Moreover, the current trend in modern supercomputer architectures is to use GPUs as low-power consumption devices. Therefore, for us it is important to analyze possible application of CUDA as a GPU programming language to speedup optimization and provide a basis for future similar applications using massively parallel systems.

B. 2-opt and 3-opt

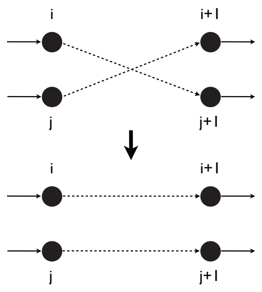

Fig. 2: A 2-opt move

The 2-opt algorithm basically removes two edges from the tour, and reconnects the two new sub-tours created. This is often referred to as a 2-opt move. There is only one way to reconnect the two sub-tours so that the tour remains valid (Figure 2). The steps is repeated only as long as the new tour is shorter. Removing and reconnecting the tour leads to an optimal route (local minimum). The 2-opt method returns local minima in polynomial time. It improves the tour by reconnecting and reversing the order of subtour. Every pair of edges (for example $[i, j+1]$ and $[j, i+1]$) is checked if an improvement is possible $((i, i+1)+(j, j+1) < (i, j+1)+(j, i+1))$. The procedure is repeated until no further improvement can be done. The 3-opt algorithm works in a similar fashion, but instead of removing two edges, three are reconnected (Figure 3). This means that there are two ways of reconnecting the three sub-tours into a valid tour + the connections being identical to single 2-opt moves. A 3-opt move can actually be seen as two or three 2-opt moves. If a tour is 3-optimal it is also 2-optimal[17]. A 3-opt exchange provides better solutions, but it is significantly slower ($O(n^3)$ complexity).

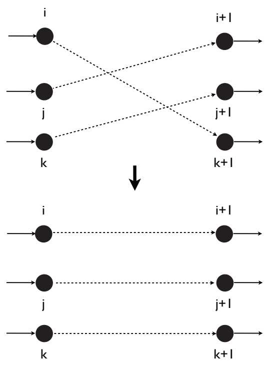

Fig. 3: A 3-opt move

II. Related works

The factorial algorithm's complexity motivated the research in two ways: exact algorithms or heuristics algorithms. The exact algorithms search for an optimal solution through the use of branch-and-bound, linear programming or branch-and-bound plus cut based on linear programming [14] techniques. Heuristics solutions are approximation algorithms that reach an approximate solution (close to the optimal) in a time fraction of the exact algorithm. TPS heuristics algorithms might be based on genetic and evolutionary algorithms [13], simulated annealing [15], Tabu search, neural networks, ant systems, among others. 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. [6] describe and evaluate a parallel implementation of iterative hill climbing with random restart for determining high-quality solutions to the traveling salesman problem. That work explains parallelization and optimization the IHC algorithm for TSP so that it can reap the benefits of GPU acceleration. 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. In their approach the search is parallelized in the way that many starting points are represented by GPU threads. This makes all the threads calculate all the possible 2-opt exchanges, so even if the solution could be found in just one step, still a significant amount of time is needed to perform at least one 2-opt search (around 0.4s). This is the reason why we decided to parallelize the 2-opt search itself, leading to a strong-

```
for (int j = 1; j < cities - 1; j++)
    for (int i = j+1; i < cities; i++) {
      /* calculate the swap change of edges i and j*/
      /*remember i and j if they are better than the known
            ones*/
    }
```

Listing 1: Iterating over the possible 2-opt exchanges forming a triangular matrix

scaling algorithm. Most of the other works related to parallel TSP solvers regards evolutionary and genetic programming, such as Ant Colony Optimization (ACO) [7] or Genetic Algorithms (GA) [8].

III. Methodology

Assuming that there are N cities in a route, the number of distinct pairs of edges can be approximated by $\frac{n*(n-1)}{2}$, which means that i.e. in case of kroE100 problem from TSPLIB[9], there are 4851 swaps to be checked, 95703 in case of pr439 problem or 2857245 in case of pr2392. Listing 1 shows the sequential approach. In order to calculate the effect of the edge exchange, the distance between the cities needs to be known. It can be obtained in two ways. The first one involves calculating the distance based on the points' coordinates (i.e. Euclidean Distance - Listing 2). The second way, typically used in CPU implementations, uses Look-Up Table (LUT) with pre-calculated distances. The main disadvantage of the latter one is the memory usage (n^2 or $\frac{n^2}{2}$) as presented in Table I. In case of 3-opt local search, the number of the exchanges to be checked increases to $\frac{n*(n-1)*(n-2)}{6}$.

TABLE I: Memory needed to store coordinates and pre-calculated distances

Problem (TSPLIB)	Number of cities (points)	Memory needed for LUT (MB)	Memory needed for coordinates (kB)
kroE100	100	0.038	0.78
ch130	130	0.065	1.02
ch150	150	0.086	1.17
kroA200	200	0.15	1.56
ts225	225	0.19	1.75
pr299	299	0.34	2.34
pr439	439	0.74	3.43
rat783	783	2.34	6.12
vm1084	1084	4.48	8.47
pr2392	2392	21.8	18.69
pcb3038	3038	35.21	23.73
fl3795	3795	54.9	29.65
fnl4461	4461	75.9	34.85

A. Pararellization

1) CPU: We applied simple parallel schemes in the CPU algorithm. Since there are $\frac{n*(n-1)}{2}$ and $\frac{n*(n-1)*(n-2)}{6}$ edge exchanges to be checked in 2-opt and 3-opt respectively, the natural approach was to distribute them between the threads equally to provide good load-balancing. Therefore the first step is to calculate how many of the checks will be performed by each thread. I.e. in case of 2-opt it will be $k = \frac{n*(n-1)}{2*threads}$. Then, the treads will be assigned the ranges of swaps (i.e. [0,k], [k+1,2k],...). Since a triangular matrix is concerned, we decided to index the cells of possible swaps in order to map them to the thread ranges (Figure 4 - each thread is given equal portion of work)). Now, based on the cell id, the original matrix indices can be calculated by solving the equation $n^2+n-2*index = 0$ (because the sum of the element in the n^{th} row equals $\frac{n(n+1)}{2}$). The root can be found using the well know formula: $\frac{-b+\sqrt{b^2-4a*2c}}{2a}$. After simplifying we obtain the indices as in the GPU code (Listing 3, lines 21-22). Then, based on the index value known by a thread, we can operate on proper matrix ranges as in Figure 4. We parallelized the

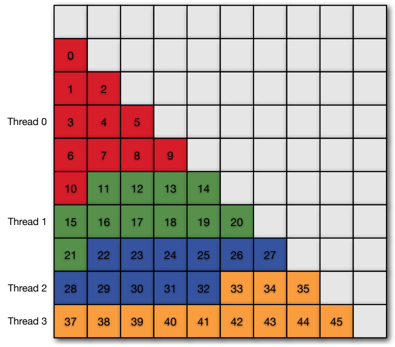

Fig. 4: Lower Triangular Matrix with cell indexing for load balancing

3-opt algorithm in the same way. The only difference is the way we indexed the swaps. The sum of the elements after each row is a tetrahedral number and the sum is defined as $\frac{n*(n+1)*(n+2)}{6}$. Therefore, we need to solve a cubic equation to distribute the work equally. The row number can be found after applying the following formula: $n = \sqrt[3]{3*index + \sqrt{9*index^2 - 1/9}} + \sqrt[3]{3*index - \sqrt{9*index^2 - 1/9}} + 1$.

2) GPU: Thousands of threads can run on a GPU simultaneously and best results are achieved when GPU is fully *loaded* to hide memory latency. The easiest approach would be to dedicate one thread to one 2-opt swap calculation. Since the large off-chip GPU memory has high latency and the fast on-chip memory is very

```
__device__ __host__int distance(int i, int j, city_coords* coords) {

    //coords stored in fast shared memory in case of GPU execution
    register float dx, dy;
    dx = coords[i].x - coords[j].x; dy = coords[i].y - coords[j].y;
    return (int)(sqrtf(dx * dx + dy * dy) + 0.5f); // + 0.5f to round
    // return Euclidean distance
}
```

Listing 2: A simple function calculating Euclidean distance for GPU or CPU

limited, we knew that accessing pre-calculated data is not a good idea. Additionally, GPU has very high peak computational power compared to CPUs. Therefore we decided to store only the coordinates of the points in the fast on-chip 48kB of shared memory (however that limits us to approximately 4800 cities[1]) and calculate the distance each time when it is needed. As in the CPU approach, we calculate first the number of checks performed by each thread: $iter = \frac{n*(n-1)}{2*blocks*threads}$. Then each thread checks assigned cell number and then jumps *blocks*threads* distance *iter* times. This allows us to avoid using global memory as it is accessed only one time at the beginning of the execution (Figure 5). I.e. For a 28 x 1024 configuration (CUDA blocks x threads) and pr2392 problem, $ceil(2857245/(28*1024)) = 100$ iterations will be necessary for each thread. That means that each thread will reuse previously stored data in the shared memory 99 times without having to access the slow global memory.

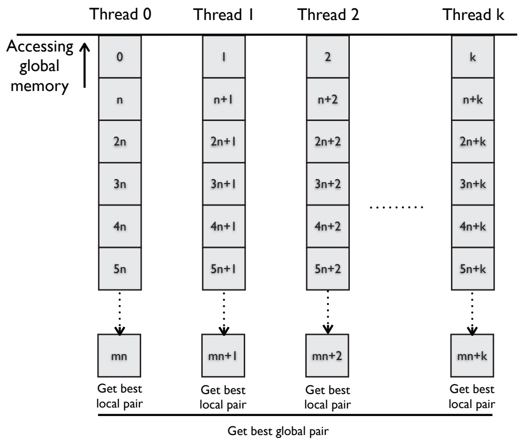

Fig. 5: GPU parallel approach, n - total number of threads, $k < n$, $mn + k \geq$ total number of checked pairs $\geq= mn$, $\{1, 2...k\}$- thread id

[1]4800 cities: 38400 B (4800 x sizeof(float) x 2) for the coordinates + 9600 B (4800 x sizeof(unsigned short)) for the route = 48000 B

IV. Results and Analysis

We tested our algorithms on 3 different platforms:

1) Tesla C2075 GPU, Intel Core i7 CPU S870, CUDA 4.1, PCIe 2.0
2) GeForce GTX 680 GPU, Intel Core i7-3960X CPU, CUDA 4.2, PCIe 3.0
3) AMD Opteron 6276 Interlagos CPU (32 cores)

We used gcc compiler with
-fomit-frame-pointer -O3 -funroll-loops -mavx -mfma4 options during CPU code compilation and nvcc compiler with *-O3 –use_fast_math* options for the GPU code.

The first table (Table II) shows a comparison of time needed to perform full 2-opt search using different TSPLIB instances and methods. CPU LUT refers to CPU Look-Up Table, a method where the distances are pre-calculated and stored in the memory, so that later, no calculation is needed. For small problem sizes this method is quite effective, at some point accessing the coordinates only and calculation become faster (possibly due to the memory throughput limitation). GPU implementations are much much faster even after including the time needed for data transfer, whose proportion to the calculation part decreases with the problem size growing (Fig. 8) It can be explained by the very fast on-chip shared memory which can be treated as an explicitly managed cache and much higher peak memory throughput compared to the CPU. Table I presented before shows how much memory is needed to store only the coordinates of the cities (2 float variables) and the pre-calcucaled distances. It is easy to understand why in case of a bigger problem instances the advantage of using a Lookup Table is diminished. We believe that CPU is reaching its memory bandwidth limits. Additionally, due to large data arrays being accessed randomly, cache efficiency is decreased drastically. This is especially visible in the parallel CPU implementation (Table III). The LUT approach does not scale beyond the factor of 4 and 9 for the 2-opt and 3-opt search respectively. While for small problem sizes it is still effective, it is better to use CPU ALU units instead of memory for more than 1084 points in 2-opt and 439 points in 3-opt. The

```
__global__ void kernel2opt(city_coords* coords, int cities, unsigned short* tour, unsigned long counter, unsigned int iter) {

  register int local_id = threadIdx.x + blockIdx.x * blockDim.x;
  register int packSize = blockDim.x*gridDim.x;
  register int i, j, change, id;
  register unsigned long max = counter;

  __shared__ city_coords c[MAX_CITIES]; //coordinates (2 x float)
  __shared__ short t[MAX_CITIES]; //current (initial tour)
  __shared__ int cit; //number of cities

   for (int k = threadIdx.x; k < cit; k += blockDim.x) { //copy current tour and coordinates to fast shared memory
      t[k] = tour[k]; c[k] = coords[k];
   }
   cit = cities;
   __syncthreads();

    for (register int no = 0; no < iter; no++) {// iter - number of iterations performed by each thread
      id = local_id + no*packSize;// calculate current 2-opt exchange index (local_id + offset)
      if (id < max) { //max - total number of 2-opt checks
          i = int(3+sqrtf(8.0f*(float)id+1.0f))/2; // get indices of the triangular matrix based on the sequential number
          j = id - (i-2)*(i-1)/2 + 1;
          // calculate 2-opt exchange's effect
          change = distance(t[j], t[i], c) + distance(t[i-1], t[j-1], c) - distance(t[i-1], t[i], c) - distance(t[j-1], t[j], c);
          if (change < local_min_change) { /*remember the values*/ }
      }
  }
  //check which thread has the lowest local_min_change and write the best values (i, j) to the global memory
  //using atomic operations
}
```

Listing 3: 2-opt CUDA kernel with shared memory reuse (pseudo code)

TABLE II: 2-opt and 3-opt time needed for a single run, GPU: GTX680, CPU: Opteron 6276, 1 core

Problem (TSPLIB)	Number of cities (points)	GPU kernel time 2-opt	GPU kernel time 3-opt	Host to device copy time	Device to host copy time	GPU total time 2-opt	GPU total time 3-opt	CPU time 2-opt	CPU time LUT 2-opt	CPU time 3-opt	CPU time LUT 3-opt
kroE100	100	31 μs	56 μs	7 μs	12 μs	50 μs	75 μs	343.7 μs	182 μs	12.6 ms	4.43 ms
ch130	130	34 μs	78 μs	7 μs	12 μs	53 μs	97 μs	498.3 μs	261.9 μs	21.7 ms	9.8 ms
ch150	150	34 μs	116 μs	7 μs	12 μs	53 μs	135 μs	602.3 μs	298.2 μs	30.6 ms	14.4 ms
kroA200	200	33 μs	219 μs	7 μs	12 μs	52 μs	238 μs	904.2 μs	419.2 μs	57.7 ms	26.7 ms
ts225	225	31 μs	276 μs	7 μs	12 μs	50 μs	295 μs	1.09 ms	491.2 μs	82.7 ms	30.5 ms
pr299	299	32 μs	573 μs	8 μs	12 μs	52 μs	593 μs	1.75 ms	750.5 μs	197.1 ms	66.1 ms
pr439	439	36 μs	1.42 ms	8 μs	12 μs	56 μs	1.44 ms	3.4 ms	1.32 ms	632.8 ms	190 ms
rat783	783	56 μs	6.67 ms	8 μs	12 μs	76 μs	6.69 ms	8.39 ms	4.4 ms	3.63 s	1.05 s
vm1084	1084	79 μs	16.95 ms	8 μs	12 μs	99 μs	16.96 ms	14.3 ms	8.28 ms	9.65 s	3.06 s
pr2392	2392	267 μs	159.9 ms	10 μs	12 μs	289 μs	160 ms	57.7 ms	62.6 ms	104.39 s	59.28 s
pcb3038	3038	413 μs	315.6 ms	11 μs	12 μs	436 μs	315.7 ms	88.8 ms	110.5 ms	214.65 s	144.16 s
fl3795	3795	628 μs	598.9 ms	11 μs	12 μs	651 μs	598.9 ms	138.9 ms	183.4 ms	420.39 s	313.33 s
fnl4461	4461	855 μs	925.8 ms	11 μs	12 μs	878 μs	925.3 ms	187.6 ms	262.08 ms	682.91 s	533.15 s

effect of memory/cache limitation can be also seen in Fig. 7. Figure compared the results of our CPU and GPU parallel implementations depending on the problem size. The overall GPU (GTX 680) speedup compared to the 32 CPU cores that we have noticed ranged from 3 to 20 and 12-26 for the 2-opt and 3-opt search respectively. Figure 9 shows the performance of the algorithm in terms of FLoating OPerations per Second (FLOPS). We recorded the peak GPU performance of 1.53 TFLOPS (single precision, approximately 50% efficiency) in 3-opt and 407 GFLOPS in 2-opt. Considering the fact that CPU parallel implementation was about 25 times slower, it would mean that CPU operated at the speed of approximately 60 GFLOPS. We would expect even higher numbers for local search like 4-opt or k-opt as the computation/memory load ratio would increase. It is also the reason why 3-opt's GPU performance is relatively higher than expected compared to 2-opt.

TABLE III: 32 CPU cores (Opteron 6276) working in parallel compared to our GPU (GTX680) implementation: times and speedups (SU)

Problem (TSPLIB)	CPU 2-opt	CPU 2-opt LUT	CPU 3-opt	CPU 3-opt LUT	CPU 2-opt	CPU 2-opt LUT	CPU 3-opt	CPU 3-opt LUT	GPU 2-opt vs CPU	GPU 2-opt vs CPU LUT	GPU 3-opt vs CPU	GPU 3-opt vs CPU LUT
	time	time	time	time	SU	SU	SU	SU	SU	SU	SU	SU
kroE100	216.8 μs	151.6 μs	1.71 ms	0.94 ms	1.58	1.2	7.37	4.71	4.34	3.03	22.8	12.5
ch130	317.5 μs	190.3 μs	2.74 ms	1.43 ms	1.57	1.37	7.97	6.85	5.99	3.59	28.24	14.74
ch150	364.1 μs	215.1 μs	3.61 ms	1.78 ms	1.65	1.39	8.47	8.09	6.86	4.05	26.74	13.18
kroA200	501.5 μs	278.5 μs	6.85 ms	3.13 ms	1.8	1.51	8.42	8.53	9.64	5.36	25.55	13.15
ts225	570.1 μs	311.6 μs	7.77 ms	3.5 ms	1.91	1.57	10.64	8.71	11.4	6.23	26.33	11.86
pr299	798.3 μs	416.2 μs	14.75 ms	8.69 ms	2.19	1.8	13.36	7.61	15.4	8.0	24.87	14.65
pr439	1.24 ms	657.1 μs	29.75 ms	35.78 ms	2.74	2.32	21.27	5.31	22.1	11.7	20.65	24.84
rat783	2.12 ms	1.53 ms	115.7 ms	0.20 s	3.95	2.88	31.37	5.14	27.9	20.1	17.29	30.49
vm1084	2.88 ms	2.49 ms	317.7 ms	0.55 s	4.96	3.32	30.37	5.56	29.1	25.2	18.73	32.42
pr2392	8.26 ms	16.33 ms	3.72 s	7.17 s	6.98	3.83	28.06	8.26	28.6	56.5	23.25	44.81
pcb3038	12.25 ms	31.62 ms	7.66 s	19.1 s	7.24	3.49	28.02	7.54	28.1	72.5	24.26	60.5
fl3795	15.23 ms	57.19 ms	15.09 s	44.96 s	9.12	3.20	27.85	6.96	25.4	87.8	25.19	75.07
fnl4461	18.70 ms	85.78 ms	24.43 s	104.5 s	10.03	3.05	27.95	5.10	20.2	97.7	26.4	112.93

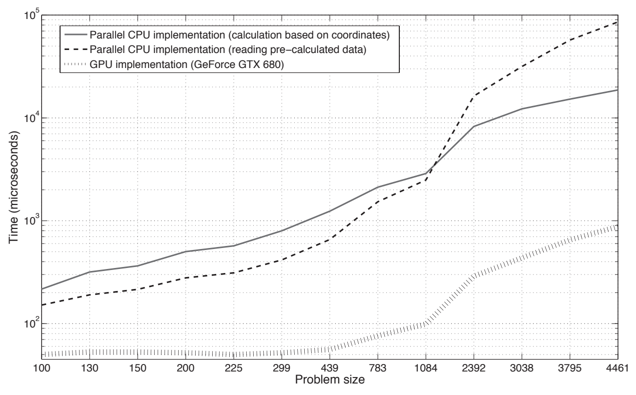

Fig. 6: 2-opt best swap search: Comparison of parallel methods used (GTX680, Opteron 6276 32 cores)

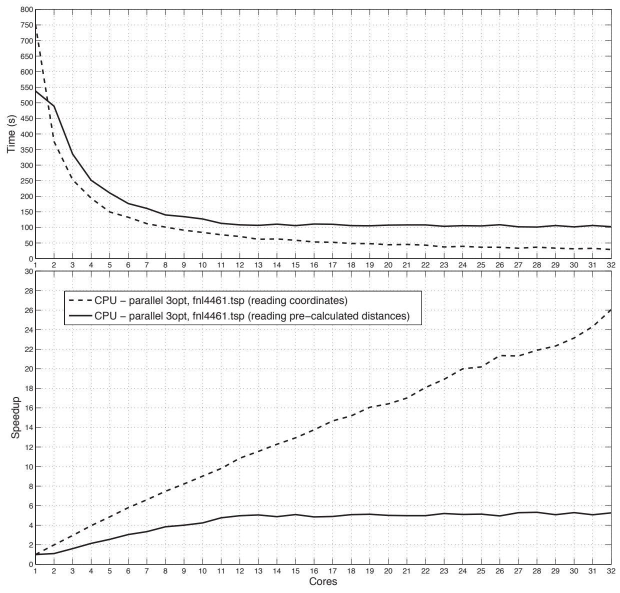

Fig. 7: 3-opt best swap search - fnl4461.tsp: Parallel CPU implementation time and speedup (Opteron 6276, 32 cores))

V. Conclusion

In this paper we have presented high-performance GPU implementations of the 2-opt and 3-opt algorithms used to solve the Traveling Salesman Problem. We used 13 TSPLIB problem instances with sizes ranging from 100 to 4461 cities. Our results show that by using our algorithm for GPU, the search time can be decreased up to 26 times compared to a 32-core CPU or over 500 times when the sequential algorithm is considered. One of our main observations is the changing programming scheme needed to take advantage of modern, highly parallel architectures. The number of processing units constantly increases, but the memory limitations remain almost the same. Therefore, in order to achieve good results with GPU or multi-core CPU, we needed to take utilize computational resources as much as possible, treating memory as a scarce resource. We believe that this approach will be proper for the upcoming new hardware as well. The next step in our research is to test the algorithm within a whole iterative search application and compare the results to our previous ones. We will try to divide the 2-opt and 3-opt problem into sub-problems to be able to solve larger instances. The main problem that we can observe right now is the limited size of the GPU's shared memory which does not allow us to store more than 4800 cities' coordinates.

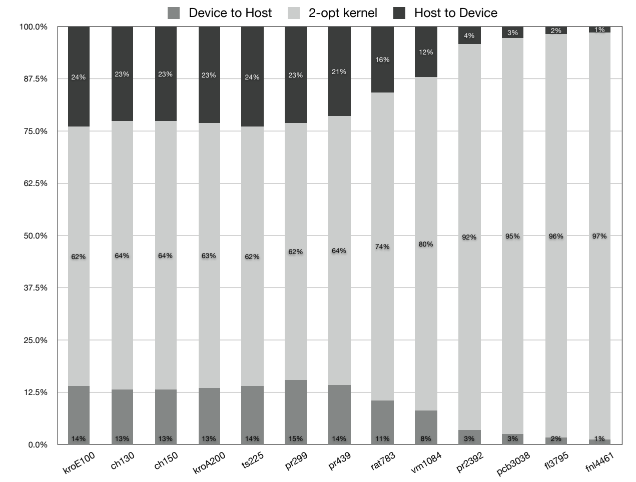

Fig. 8: 2-opt GPU search: Proportion of kernel execution to the communication time (GTX 680)

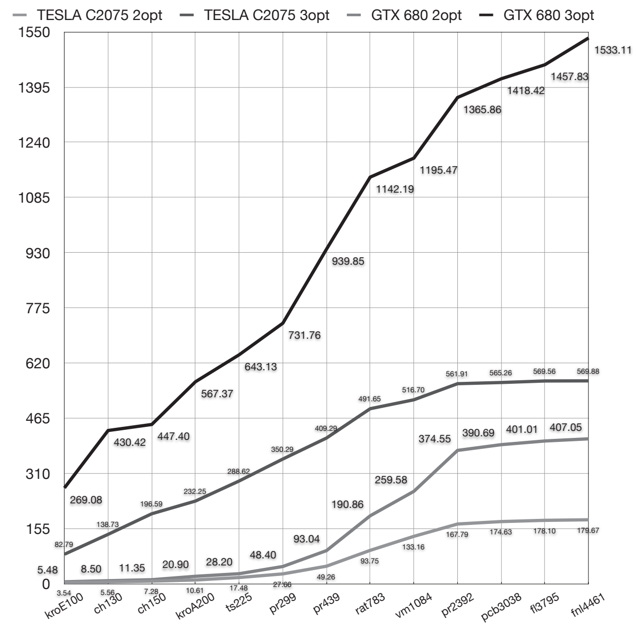

Fig. 9: GPU: GFLOPS observed during single 2-opt and 3-opt kernel executions (GTX 680, TESLA C2075)

ACKNOWLEDGMENT

This work was supported by Core Research of Evolutional Science and Technology (CREST) project of Japan Science and Technology Agency (JST) and Grant-in-Aid for Scientific Research of MEXT Japan.

REFERENCES

[1] Croes G. A.;A Method for Solving Traveling-Salesman Problems, Operations Research November/December 1958 6:791-812;

[2] 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.

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

[4] 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)

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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

[12] S. Baluja, A.G. Barto, K. D. Boese, J. Boyan, W. Buntine, T. Carson, R. Caruana, D. J. Cook, S. Davies, T. Dean, T. G. Dietterich, P. J. Gmytrasiewicz, S. Hazlehurst, R. Impagliazzo, A.K. Jagota, K. E. Kim, A. Mcgovern, R. Moll, A. W. Moore, L. Sanchis, L. Su, C. Tseng, K. Tumer, X. Wang, D. H. Wolpert, Justin Boyan, Wray Buntine, Arun Jagota: Statistical Machine Learning for Large-Scale Optimization, 2000

[13] Tsai, H.; Yang, J. Kao, C. Solving traveling salesman problems by combining global and local search mechanisms, Proceedings of the 2002 Congress on Evolutionary Computation (CEC'02), Vol.2, pp. 1290-1295.

[14] Karp, R. Reducibility among combinatorial problems: In Complexity of Computer Computations. Plenum Press, pp. 85-103. New York, 1972

[15] Pepper J.; Golden, B. Wasil, E. Solving the travelling salesman problem with annealing-based heuristics: a computational study. IEEE Transactions on Man and Cybernetics Systems, Part A, Vol. 32, No.1, pp. 72-77, 2002

[16] NVIDIA CUDA Programming Guide
http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf

[17] Helsgaun, K.; An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic, European Journal of Operational Research, 2000, vol 126, pages 106-130

[18] Nilsson, Ch.; Heuristics for the Traveling Salesman Problem, Linkoping University, pages 1-6