JURAJ FOSIN, Ph.D. Student
E-mail: juraj.fosin@fpz.hr
University of Zagreb,
Faculty of Transport and Traffic Sciences
Vukelićeva 4, 10000 Zagreb, Croatia
DAVOR DAVIDOVIĆ, Ph.D. Student
E-mail: davor.davidovic@irb.hr
Ruđer Bošković Institute
Bijenička cesta 54, HR-10000, Zagreb, Republic of Croatia
TONČI CARIĆ, Ph.D.
E-mail: tonci.caric@fpz.hr
University of Zagreb,
Faculty of Transport and Traffic Sciences
Vukelićeva 4, 10000 Zagreb, Croatia

Transport Engineering
Preliminary Communication
Accepted: June 12, 2012
Approved: May 23, 2013

A GPU IMPLEMENTATION OF LOCAL SEARCH OPERATORS FOR SYMMETRIC TRAVELLING SALESMAN PROBLEM

ABSTRACT

The Travelling Salesman Problem (TSP) is one of the most studied combinatorial optimization problem which is significant in many practical applications in transportation field. The TSP is an NP-hard problem and requires large computational power to be optimally solved by exact algorithms. In the past few years, fast development of general-purpose Graphics Processing Units (GPUs) has brought huge improvement in decreasing the algorithms execution time. In this paper, we implement 2-opt and 3-opt local search operators for solving the TSP on the GPU using its respective application programming interface. The novelty presented in this paper is a new parallel iterated local search implementation with 2-opt and 3-opt operators for symmetric TSP, optimized for the execution on GPUs. With our implementation large TSP problems (up to 85,900 cities) can be solved using the GPU. We show that our GPU implementation can be up to 27 times faster than central processing unit (CPU) implementation without losing solution quality for TSPlib problems as well as for our CRO TSP problem.

KEY WORDS

travelling salesman problem, local search operator, 3-opt, parallel iterated local search, graphics processing units, CUDA

1. INTRODUCTION

The travelling salesman problem (TSP) is a well-known combinatorial optimization problem which is important in many practical applications from various fields such as transport, electronics and other engineering fields. The goal of solving the TSP is to find the tour of the minimal cost that the salesman can take. He has to visit each city from the list of n cities exactly once and then return to the home city. The cost of the travelling from any city i to any other city j is a known value and is stored in the two-dimensional matrix C, where c_{ij} represents the cost of going from city i to city j ($i,j = 1,\ldots,n$). A possible solution (tour) is represented as an n-size vector $(i_1, i_2, i_3, \ldots, i_n)$, where i_k is the city i on the k-th position in the tour. To find the optimal tour we have to find a vector $(i_1, i_2, i_3, \ldots, i_n)$ that minimizes the cost function

$$f(i_1, i_2, i_3, \ldots, i_n) = c_{i1i2} + c_{i2i3} + \ldots + c_{ini1}.$$

In TSP we are looking for a Hamiltonian tour of minimal length on a fully connected graph. If the distance between two cities in both directions is the same, then the problem is called a symmetric travelling salesman problem (STSP). The TSP is a difficult problem and belongs to the *NP*-hard class of problems [1] where the worst-case running time of exact algorithm grows faster than any polynomial. As the size of the fully connected graph grows, the feasible solution space size raises as factorial. Therefore, an exhaustive search algorithm is impractical and heuristic methods are used to speed up the process of finding a satisfactory good solution. Although the heuristic methods decrease the search space of feasible solutions, they still require a large amount of computational resources.

A new and affordable source of computing power that can be used to solve this type of problem is a graphics processing unit (GPU). In the past the GPUs were designed specifically for graphic rendering and

thus were very restrictive in terms of operations and programming, but today that has changed drastically. The term general-purpose computing on graphics processing units (GPGPU) is commonly used when GPU is used to perform general-purpose computation that was traditionally handled by a central processing unit (CPU). Although programming languages and frameworks that use GPUs for non-graphical, computationally intensive operations, were known for a longer time (i.e. Brook GPU [2]), real ascent appeared at the end of 2006, when both major GPU companies released their APIs (application programming interface), Nvidia CUDA [3] and ATI Stream [4]. Later, Microsoft's Direct Compute and Open CL by Khronos Compute Working Group were published. Theoretically, the GPUs can perform much more floating point operations per second (flops) and have larger bandwidth rate compared to the CPUs, but on the other hand, GPUs cannot perform multiple tasks in parallel. They are capable of running a same task on the different data concurrently and because of that not all problems can fit the GPUs paradigm and exploit their superior performance.

In this paper, a new parallel iterated local search approach for solving TSP with parallel 2-opt and 3-opt operators optimized for the execution on GPUs is presented. Using the best improvement search step selection mechanisms, the operator's parallel execution is fully achieved and the problem of warp divergences (different threads in a warp take different paths that are leading to serialisation) is overcome. The biggest achievement of this research is that we have developed the 2-opt and 3-opt local search, executed on the GPU that can solve large problems (e.g. the biggest TSPlib problem with 85,900 cities). To the best of our knowledge, this is the first GPU algorithm able to solve the biggest TSPlib problems.

The rest of the paper is organized as follows: In Section 2 heuristic algorithms for solving TSP problem are summarized as well as some speed-up techniques. In Section 3 NVIDIA GPU architecture is briefly described. Section 4 is the main part of this paper where the implementation details of 2-opt and 3-opt local search on the GPU are described. In Section 5 the test results are presented, and finally the conclusion is made in Section 6.

2. HEURISTICS FOR SOLVING TSP

Comprehensive surveys of algorithms used for solving TSP can be found in [5, 6, 7, 8]. Many efficient methods for solving TSP are two-phase algorithms. In the first phase, the initial tour is generated by a constructive algorithm. In the second phase, the local search is involved for finding the local optimum. When the local optimum is found the escape mechanism should be activated in order to reach a new global optimum. The algorithm stops after reaching some stopping criteria such as the given execution time, number of iterations, quality of solution or a combination thereof. The constructive algorithms for initial solution that are widely used for TSP are the Nearest Neighbour Heuristic (NNH), Greedy and Boruvka. Bentley [9] noticed that a better starting tour provides better final results. The Greedy heuristic provides better final results for 2-opt and 3-opt than any other known [7].

Among many approaches for the local search in solving TSP, the *k*-opt has the strongest impact. This approach converts one feasible route to another and performs conversions until the reduction of the length of the tour is possible. The conversion is done by removing and adding the edges. In 2-opt, a special case of the *k*-opt, the two edges are deleted, and the opposite ends are joined together [10]. The 3-opt considers deleting three edges and finding the best reconnection combination [11]. With the simple 3-opt it is possible to reach closer than 5% of the optimal solution [1].

The *variable*-opt approach used by Lin and Kernighan [12] yields results around 2% above the optimal solution [1]. The latest significant improvement in reaching optimal or near-optimal solutions is done again by *k*-opt sub-moves approach by Helsgaun [13] for the problems with a size from 10,000 to 10,000,000 cities.

Moreover, there are numerous techniques and mechanisms that can improve local search and avoid some common problems (i.e. reaching local optimum). One of them is double bridge move or 4-opt move that was first mentioned in [12] as an example of simple move which cannot be normally generated by 3-opt or Lin-Kernighan algorithm. Many modern algorithms use this move for the escape from the local optima.

Until now there have been just a few articles that investigated the possibilities of using the GPU graphic card in solving the problems defined on a graph such as TSP [14, 15]. Prior works on GPU-based TSP solver were based on some variants of Ant Colony Optimization (ACO) [16] which has been inspired by ant foraging behaviour that results in finding the shortest path between their nest and the food source. Among others, CUDA implementation of ACO is proposed by Bai et al. [17], You [18], Lin et al. [19] and Celilia et al. [20]. The focus of most of them is on the comparison of GPU and CPU implementation and they do not address how often their implementation reaches the optimal TSP solutions. To the best of our knowledge, there have been a few other articles that investigated the possibilities of using the GPU graphic card in solving TSP such as the evolutionary algorithm [15] and immune algorithm [14]. Even the study that uses CUDA implementation of hill climbing [21] or the one in which a parallel local search is used [22] fail to run GPU implementation on TSPLIB problems which have more than 2,000 cities. With our implementation of TSP solver

much larger problems on the GPU can be solved. The largest problem we have solved so far was 85,900 cities but we are also experimenting on the problem with 200,000 cities.

2.1 Techniques for search space reduction

The TSP search space is very large because all the permutations of the cities, where each city appears exactly once, are valid solutions and there is no constraint that reduces the potential sets of permutations. "Naive" implementation of 2-opt and 3-opt operators needs to examine n^2 pairs or n^3 triples in each search step. Over the years of comprehensive scientific research on TSP, various speed-up techniques were applied. Johnson and McGeoch [7] pointed out four techniques as the key ones: avoiding search space redundancy, bounded neighbour list, *don't look bits* (DLB), and tree-based tour representation.

Avoiding search space redundancy is related to the observation by Lin and Kernighan [12]. The edge to be added should not be longer than the corresponding edge to be deleted by the operator. More formally, city c_j will not be considered unless $c_{i,j} < c_{i,i+1}$. This rule can be used for 2-opt and 3-opt operators. Bounded neighbour list assumes that the best solutions do not contain long edges, so each city c_i needs a sorted bounded list of its neighbour cities to be evaluated for 2-opt or 3-opt move. The common neighbour list contains 10 to 40 neighbours. The nearest neighbour to the city c_i is the city c_j for which $c_{i,j}$ is minimal. There are several ways of creating the neighbours list. The one utilized in our research is the so-called quadrant-nearest neighbour [7]. This list contains equal number of the nearest neighbours from every quadrant (with the centre in the observed city). Helsgaun proposed more complex construction of the list, based on α-values [23]. The α-nearest neighbour list better covers the edges in the optimal solutions than the classical approaches and therefore can be much smaller. For instance, Helsgaun uses only 5 neighbours for every city.

DLB was introduced by Bentley [9] to avoid unnecessary repetitions of the local search for improving moves. Bentley proposed to exclude cities from the search if previously no improving move for them was found. For each city a single bit is allocated, which is at the beginning switched off (set to zero), and if search finds no improving move for the respective city, a "*don't look bit*" is switched on (set to one). At the end of the local search all DLBs should be switched on, bringing additional advantage if DLB is used in iterative local search. After performing one double bridge move to get a new starting tour, DLB for vertices at the end of deleted edges are switched on. This means that local search starts with only 8 DLBs off, instead of all DLBs off, which has a great impact on the running time of the algorithm.

The tree-based tour representation speeds up the time spent on performing the calculated moves. Bentley [9] observed that as the number of cities increases, the tour changing dominates in the overall algorithm running time if the tour is implemented as an array or double-linked list. Very fast implementation of local search and very large-scale TSP problems are assumed.

Another important method to speed up the local search is to select the next search step more efficiently, which results in the significant decreasing of the algorithm execution time. The most widely used search step selection mechanisms are the best improvement and the first improvement [8]. The main difference between them is that the best improvement needs to finish evaluation of all neighbours in each search step and then apply the best move, while the first improvement immediately performs the step after the first improving move is encountered. State-of-the-art algorithms usually use first improvement approach, since only a small portion of the neighbourhood needs to be evaluated.

3. CUDA ARCHITECTURE

For the purpose of understanding the causes of the GPU performance issues in our work and to be able to understand the rest of the paper, some basic description of the GPU architecture and programming model is required. The easiest way to explain the GPU architecture and GPGPU programming model is to give side-by-side difference compared to the well-known CPU architecture and programming model.

GPGPU stands for General-Purpose computing on the GPU, also known as GPU Computing. GPUs are capable of very high computational and data throughput. In the past, GPUs were specially designed for computer graphics and were difficult to program. Today, GPUs have evolved in general-purpose parallel processors and are easier for programming [24]. The speed-ups achieved by using GPUs are of order of magnitude compared to optimized CPU implementations.

The main difference between CPU and GPU computing lies in the fact that the GPU consists of a large number of lightweight stream processors (joined in the streaming multiprocessors) that are capable of performing many relatively simple calculations in parallel.

The CPU consists of a smaller number of computational units (2-8), a large control unit and cache (on-chip) memory, while the GPU consists of a large number (32-448) of lightweight processors that share common small control unit and cache. This type of architecture is called the stream-based processing model which depends on the fast data throughput, unlike

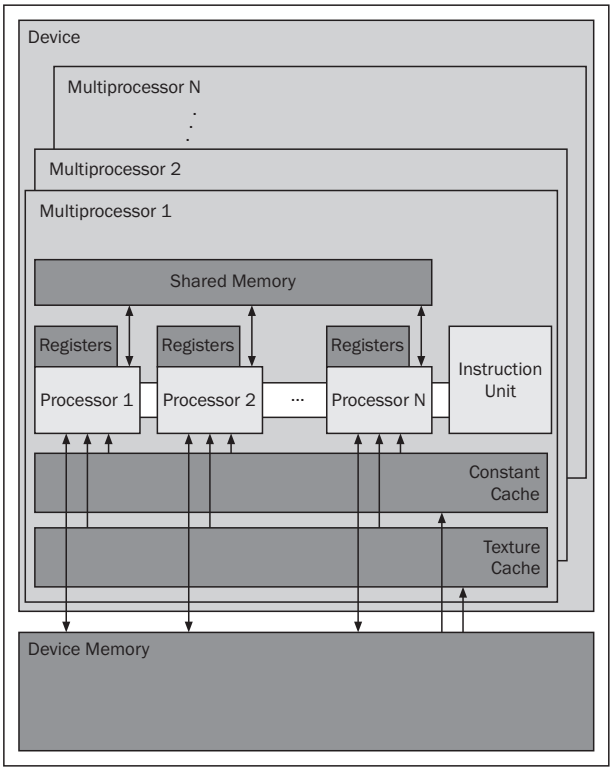

Figure 1 - GPU memory hierarchy

Source: NVIDIA CUDA C Programming guide version 3.1.1 [24]

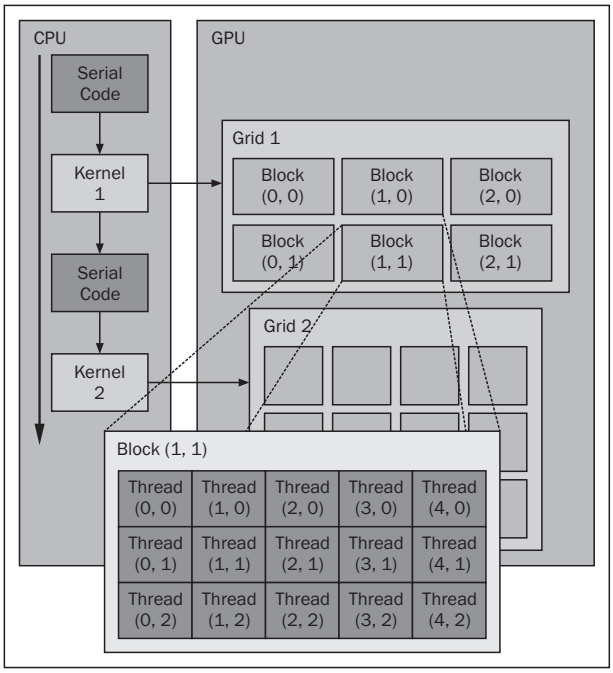

Figure 2 - CUDA Programming model

Source: NVIDIA CUDA C Programming guide version 3.1.1 [24]

the CPU that relies on the large instruction throughput. This is the reason why the GPUs are perfectly well-suited to address the problems that can be expressed as data-parallel computations, with high arithmetic intensity (the ratio of arithmetic operations to memory operations).

Figure 1 presents the memory hierarchy of the GPU that consists of three levels: first is the global (device) memory that is accessible by all stream processors, the second level is the Shared Memory on each stream multiprocessor, and the third level memory is the local memory (Register) owned by a stream processor. A GPU device has large device memory that is accessible by all multiprocessors and stream processors. Every multiprocessor consists of the on-chip share memory with low latency and high bandwidth that is shared among lightweight stream processors. Every stream processor has its own register. With careful programming, taking into account memory hierarchy, one can significantly improve overall performance of the program that is executed on the GPU.

In *Figure 2* the CUDA programming model is presented. The smallest execution unit that can be executed on one stream processor is called thread. Threads are combined into equally sized blocks, each containing up to 1,024 threads. All blocks are grouped in the grid that represents one kernel (function) that is called from the CPU and executed on the GPU. During the execution, blocks are referenced by (i, j) position in the Grid and then they are dynamically distributed to the streaming multiprocessors to be processed. A maximum of 32 threads from the same block can be run in parallel and this set is called a warp.

4. IMPLEMENTATION OF LOCAL SEARCH OPERATORS

In this section the implementation of the local search on the GPU is described. The search space is bounded with quadrant neighbour list to avoid redundancy. The quadrant neighbour list is generated on the CPU in pre-processing procedure and stored on hard drive for later reuse. Since we had no intent to use benchmarks containing more cities than the greatest TSPlib problem, the tour is represented as a linked list. We have chosen to use the best improvement search step selection mechanisms, because of the practical reasons. The problem with the implementation of the first improvement on GPU lies in the fact that when one thread finds an improvement it has to send a message to all other threads to stop their further execution. It requires the usage of thread synchronization mechanisms between threads from different blocks that, because of the specific architecture of the graphic cards, results in increasing of the execution time. Also, our experiments showed that *don't look bits* do not work well with the best improvement selection mechanism. The reason is that the local search is trapped too fast in a local optimum and thus results are significantly worse, and in addition, most of the threads do not perform any calculations, but wait for other threads to fin-

ish. Furthermore, tests showed that it is faster to keep the coordinates of the cities in the GPU main memory and calculate the distance at the GPU run-time than to read the saved distance from the GPU memory. The reason for that lies in the latency. Reading data for each thread from GPU memory requires more time than performing calculations.

For the initial tour construction, we have chosen the nearest neighbour heuristic (NNH). The NNH builds the initial tour by choosing random city as the first city in the tour and then adding the closest city (from the quadrant neighbour list) at the end of the tour, until the tour contains all cities. The number of neighbours is empirically determined to $m = 40$ for all tests presented in this paper.

The local search is the slowest part of the proposed algorithm, and thus 2-opt and 3-opt operators are the best candidates to be implemented on GPU. Generally, the 2-opt and 3-opt local search pass through all possible pairs/triplets and for each pair/triplet calculate the savings for the new tour, find the global best improvement, and perform the move (apply the best found improvement to the current solution) on the CPU.

In particular, the 2-opt and 3-opt are run in steps/iterations. At the beginning we start with the initial tour (generated in the pre-processing phase). The coordinates of each city and the neighbour table (list) are stored in the GPU global (device) memory and have the same lifetime as the algorithm. At the beginning of each iteration the current best tour is copied to the GPU and the improving moves are calculated on the GPU for each pair/triplet. The list of the improving moves are then returned to the CPU where the best improving move is calculated and applied to create a new current best (i.e. shorter) tour. The iterations are performed until the local minimum is found or a termination criterion is met.

As described in the previous section, threads are grouped in blocks (up to 1,024 threads per block) and all threads within one block can share the same memory (shared memory of the block). Since the data retrieval from the GPU main memory is much slower than the computation, we want to take advantage of the shared memory and reuse data once loaded in the block-shared memory. Therefore, we have to find the optimal number of threads per block *tb*, regarding the available shared memory. We experimentally determined that $tb = 128$ is the best number for our test-bed GPU. Once the number of threads per block is determined, the number of blocks *nb* is calculated as follows:

$$nb = \left\lfloor \frac{n * m + tb - 1}{tb} \right\rfloor$$

where *m* is the number of neighbours and *n* is the total number of the cities. For example, for the TSPlib problem d18512.tsp [25] the total number of threads is 740,480 and the number of blocks is 5,785.

4.1 2-opt local search

The 2-opt operator is the simplest and easiest to implement of all operators in the *k*-opt family for solving STSP. In 2-opt operator, the two edges are deleted, and the opposite ends are joined together, see *Figure 3* left. The input for 2-opt operator is a pair of cities (i,j). The current best tour is represented as the linked list where each list-element represents a city and the position in the list represents the position in the tour. Therefore, the first elements to the left and to the right of the current element/city represent the neighbourhood cities in the tour. The idea of the 2-opt operator local search is to pass through all possible pairs of the cities and to find a pair whose elements are not connected in the current tour (whose elements are not adjacent in the list). If such pair (i,j) is found, a new connections/paths are examined as described in *Figure 3*, left. After applying the improvement, one pass of the 2-opt local search is finished and the process is repeated until no new pairs that can decrease the total length of the tour can be found. For the symmetric problem with *n* cities the total number of pairs that have to be checked is $n(n-2)/2$. In order to decrease the number of pairs that have to be evaluated only the m nearest neighbour cities for each city are observed. It showed that the reduction of the possible pairs does not affect the total algorithm performance. Despite the search space reduction, the most time-consuming part of the 2-opt local search is still the calculation of the distance for all feasible pairs, and therefore 2-opt local search is the best candidate to be executed on the GPU.

The basic idea for our GPU implementation is to divide the 2-opt local search into small tasks in such a way that each GPU thread evaluates exactly one pair. At the end, the results from all threads are collected and the best improvement is applied on the current tour. The 2-opt operator performs the best improvement on the CPU because it is strictly sequential and cannot benefit from GPU.

The list of neighbours and the current tour are stored in the device (GPU) memory. For the performance purposes the list of neighbours remains in the device memory between kernel calls while the current tour is updated with each new kernel call. A kernel call is a function that calculates the tour length improvement for each feasible pair of elements (cities) and returns its value to the CPU. The kernel is performed on the GPU device. The test showed that the best performance is achieved if the block has $tb = 128$, where *tb* are threads per block. In order to improve the performance and to avoid the un-coalesced memory calls from the global memory, the adjacent threads of the same block examine the neighbourhood cities of the current city. The threads per block tb and the number of neighbours do not depend on each other. The variable *i* is the iterator through the current tour and *j* is the iter-

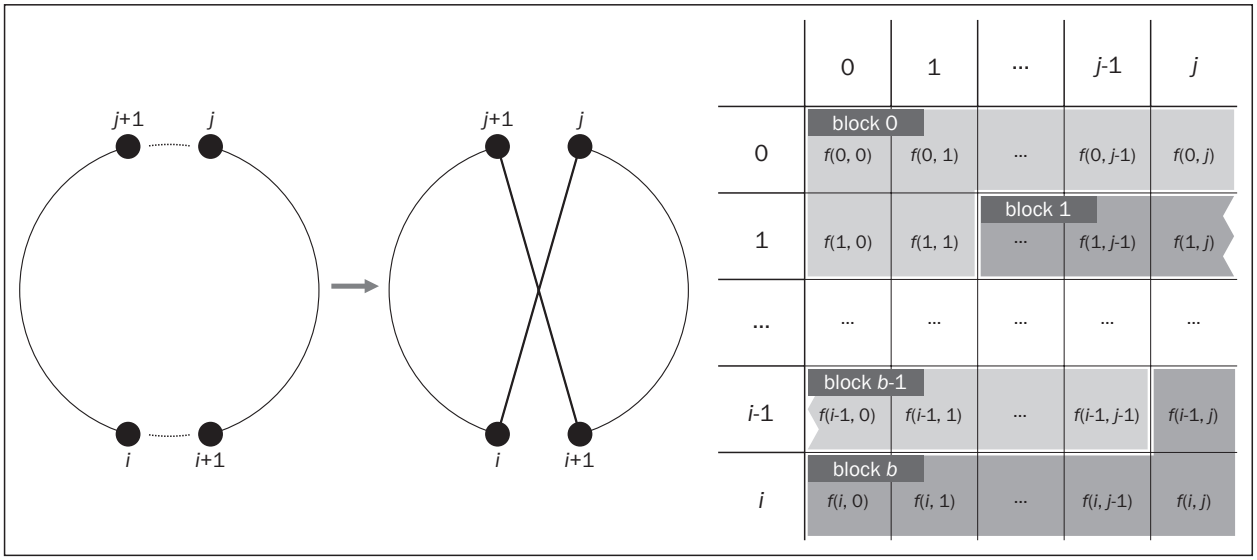

Figure 3 - left: 2-opt, right: implementation on CUDA architecture

ator of each customer's neighbours list. Each city has $j+1$ neighbours and for example each block contains $j+3$ threads, in block 0 threads $0,1,\ldots,j$ evaluate the new distances for the city pairs $(0,0),(0,1),\ldots,(0,j)$of the city c_0 and the threads $j+1,j+2$ pairs $(1,0)$and $(1,1)$ of the city c_1 (*Figure 3*, right). The $f(i,j)$ denotes the function that calculates the gain to the total tour length for the specific pair (i,j) of the cities. The first element in the (i,j) notation represents the city index and the second one the neighbour ordinal in the neighbourhood list. With this implementation it is possible to make coalesced access to the device main memory and moreover, the shared memory can be used for storing common data for all threads in the same block (in the previous example cities c_0 and c_1). It is possible that the some threads in the last block b (*Figure 3*, right) remain idle because the number of blocks *nb* is rounded to the higher whole number.

The distances between cities are calculated on-the-fly, during the kernel execution. As for the GPU architecture, it shows that calculating the distances with data already stored in the shared memory and registers is much faster than fetching data from the device main memory.

4.2 3-opt local search

In contrast to 2-opt operator, the 3-opt operator chooses the best triple (i,j,k) not yet connected in the current tour. For the 2-opt operator there is only one way to reconnect the tour fragments after deleting the two selected edges. The 3-opt operator has four different ways how to reconnect the ends after removing the set of three edges (*Figure 4*, left). Since the "naive" implementation of the 3-opt local search needs to evaluate $n(n-1)(n-2)/3$ possible changes, it is

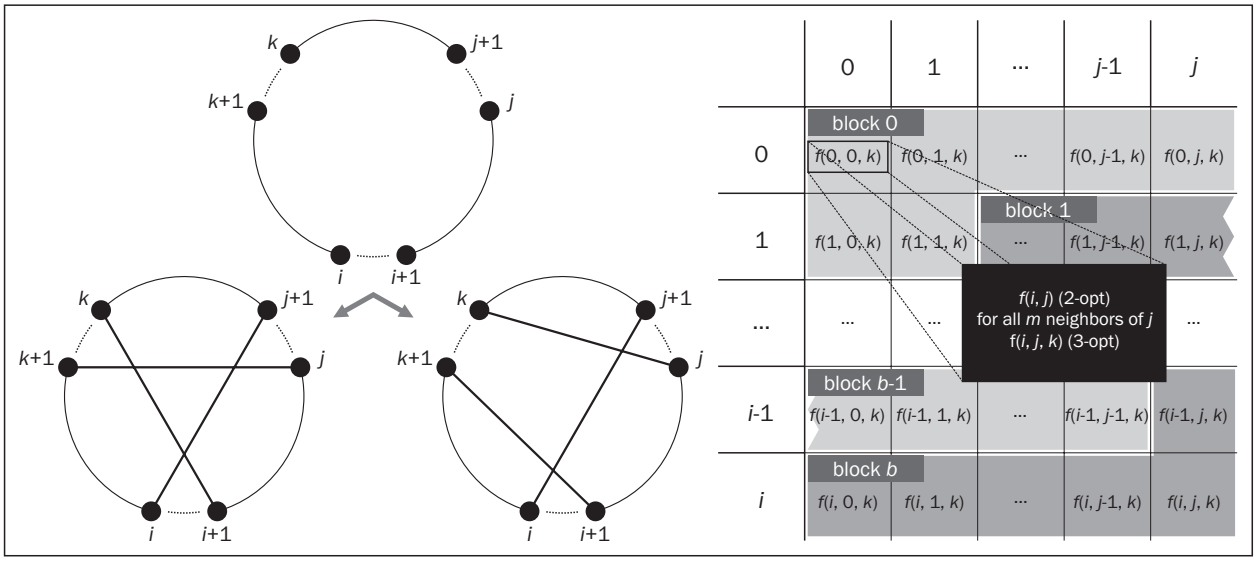

Figure 4 - left: two of four ways of reconnecting edges for 3-opt, right: implementation on CUDA architecture

obvious that regardless of GPU theoretical computing power, a significant search space reduction is needed. As in 2-opt local search, the search space reduction is done by observing only the *m* nearest neighbours for each city.

A similar idea was applied for the 3-opt local search GPU implementation as was done for the 2-opt local search. The 3-opt local search is implemented as an extended 2-opt local search, in which every thread handles a single pair (i,j) (as described for 2-opt) plus an additional loop that passes through all the neighbour cities of the city *j*, denoted as *k* (*Figure 4*, right). From the GPU implementation point of view, each GPU thread calculates the tour length improvement for triples (i,j,k), where $k = 1,\ldots,m$. In this way it is possible to exploit the distances calculated for 2-opt local search and reduce the overhead calculations in the 3-opt local search. Some mid-results calculated from other adjacent threads within the same block are stored in the GPU shared memory and reused by other threads from the same block.

4.3 Iterated local search

Iterated Local Search (ILS) for TSP has a long history, and some of the hybrid stochastic local search algorithms are among the best-performing TSP algorithms currently known [8]. The outline of the algorithm is given in *Algorithm 1*.

Algorithm 1 - Iterated local search

```
1.  init:=NNH()
2.  s:=ThreeOpt(init)
3.  best:=s
4.  while not Terminate() do
5.      s':=DoubleBridge(s)
6.      s'':=ThreeOpt(s')
7.      if f(s'')<f(best) then
8.      best:=s''
9.      endif
10. s:=s''
11  end while
```

Like in almost all ILS algorithms for the TSP problem [8], for perturbation a double-bridge move has also been chosen (*Algorithm 1*, line 5). Our implementation randomly selects four edges to be deleted and partial tours are then reconnected. As mentioned earlier, the only computationally demanding part of this ILS is the 3-opt local search (line 6) described in Section 2 and Subsection 4.2. The natural way is to perform a 3-opt local search on the GPU while the rest of the computation is performed on the CPU. The stopping function *Terminate*() (line 4) finishes the execution after having performed a certain amount of iterations. In our tests three possible numbers of iterations were selected: *0.1n*, *1n* and *10n*, where *n* is the number of cities. The number of iterations is chosen empirically with the aim of reaching high quality solutions within a reasonable time. Due to a large amount of computational time needed for CPU 3-opt local search, the ILS variant with CPU 3-opt is calculated only for *0.1n*.

5. RESULTS

In this section the results of our implementation of the 2-opt and 3-opt local search and ILS algorithms on the standard TSPlib problem library as well as proposed CRO TSP problem will be presented. Most authors use the same test-bed TSPlib set of the problems for the STSP [25] which is the library of sample instances for the STSP from various sources and for all of these problems the optimal solutions are known. This is why we have also made our computational experiments on this set of problems.

All benchmarks are performed on 32-bit Windows 7 desktop PC, running Intel i7 920 2.66 GHz processor and Nvidia GTX 470 GPU with 448 stream processors. The algorithm is tested on the standard TSPlib test instances and each instance was run 10 times. The t_{CPU} and t_{GPU} represent CPU and GPU average execution times. The speed-up is calculated by the formula: t_{CPU}/t_{GPU}, if the speed-up is greater than 1 the GPU execution time is smaller than the CPU execution time.

In Section 4 it has been described that 2-opt and 3-opt local search operators are the most time-consuming parts of the ILS algorithm. Thus, for CPU and GPU comparison of the ILS algorithm the most important information is the speed-up of the single 3-opt local search (*Table 1* and *Table 2*). A single 3-opt local search is only one call of the GPU kernel *ThreeOpt*() (*Algorithm 1*, line 6), i.e. only one pass through all pairs (i,j) is performed.

The test for a single 3-opt local search has been conducted on all Euclidian 2D TSPlib problem instances. The instances are divided in three groups in order to clarify the representation of the results. The first group includes problems between 70 and 400 cities, the second between 400 and 2,000, and the third problems group whose size is larger than 2,000 cities. The problems smaller than 70 cities are too small to be considered for the described GPU implementation due to low GPU utilization. As can be seen in *Table 1* the minimal speed-up for the first group is 0.5 and therefore for smaller problems the speed-up would be even worse. *Table 1* presents the average time of 10 runs for the evaluation of the 3-opt local search. The smallest speed-up is in the group 70-400 because the smallest problem sizes are too small to be able to fully utilize the capabilities of the GPU. As can be seen, with the increasing problem size the speed-ups (min, max and average) also increase, as was expected. The best performance is achieved for the third group (2,000-18,512) because the problem size is big enough to

Table 1 – 3-opt local search analysis for different problem group sizes, where tCPU and tGPU are average times of the average times of 10 runs for each TSPlib problem instance group

Problem group size	No. of problems	t_{CPU} [ms]	t_{GPU} [ms]	Speed-up		
				min	average	max
70-400	37	0.4371	0.2670	0.50	1.64	3-40
400-2000	25	2.6222	0.3561	3.49	7.36	13-15
2000-18512	22	17.1551	1.6981	7.46	10.10	21-07

generate the maximum number of threads to fully utilize the GPU multiprocessors. The conclusion is that the GPU implementation is faster for all TSPlib problem instances with more than 400 cities.

The average speed-up and the average execution time of 3-opt local search for the two largest TSPlib problems, pla33810 and pla85900 with 33,810 and 85,900 cities, respectively, are given in *Table 2*. It can be seen that GPU outperforms the CPU implementation even more for the largest problems.

In the next paragraphs the average speed-up and the average deviation (Δ) of the 2-opt Search and 3-opt Search algorithms and ILS algorithm from the optimal solution for Euclidian 2D TSPlib problems (up to 18512 cities) are presented. The summary of the 2-opt Search and 3-opt Search algorithms and ILS algorithm are given in *Table 3*. The 2-opt Search and 3-opt Search algorithms are similar to ILS algorithm but without the double-bridge move (*Algorithm 1*, line 6). The 2-opt Search and 3-opt Search algorithms are terminated if no better solution can be found. The stopping criterion for ILS is as described in Section 4.3. The results for the CPU version of ILS *1n* and ILS *10n* are not available (n/a) because of the extremely long execution time. The slight difference in the deviations (Δ) between CPU 2-/3-opt Search and GPU 2-/3-opt Search occurred because it is not possible to control the order of the block execution on the GPU that results in different order of application of the improvements.

Table 2– 3-opt local search analysis for the two largest TSPlib problems

	tCPU[s]	tGPU[s]	speed-up
pla33810	0.0924	0.0037	24.55
pla85900	0.2505	0.0091	27.25

As can be seen in *Table 3*, GPU outperforms CPU in speed-up in some cases more than 20 times. The minimum speed-up is low because only the small problem sizes contribute to this value. This is in line with the results in *Table 1* for the group 70-400. On the other hand, the maximum speed-up is large because only the big problems contribute to it. As expected, 3-opt Search gives smaller deviation ($\Delta_{CPU}, \Delta_{GPU}$) from optimal solutions than 2-opt Search (*Table 3*) but requires longer execution time as illustrated in *Figure 5*. The GPU 3-opt Search version outperforms both CPU 2-/3-opt Search in the execution time (*Figure 5*) and at the same time retains the Δ of the CPU 3-opt Search version (*Table 3*).

The graph comparison of ILS on both, CPU and GPU is given in *Figure 6*. The ILS algorithm significantly improves the deviation from the optimal solution (*Table 3*), but the overall time increases by a factor of approximately 10 (*Figure 6*). Furthermore, the ILS algorithm has obtained the optimal solution for 26 out of 78 TSPlib problems for *10n* iterations. The possibility of obtaining better results (more ILS iterations can be performed) greatly expands with the use

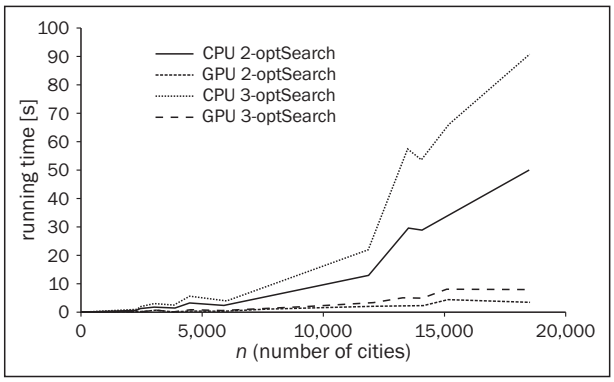

Figure 5 – Comparison of 2-optSearch and 3-optSearch running times on CPU and GPU

Table 3 – 2-optSearch, 3-optSearch and ILS summary results

	Δ_{CPU} [%]	Δ_{GPU} [%]	speed-up		
			min	average	max
2-opt Search	5.21	5.19	0.13	11.13	17.50
3-opt Search	4.58	4.52	0.50	9.98	19.28
ILS (0.1n)	2.00	2.00	0.84	8.38	20.50
ILS (1n)	n/a	0.78	n/a	n/a	n/a
ILS (10n)	n/a	0.32	n/a	n/a	n/a

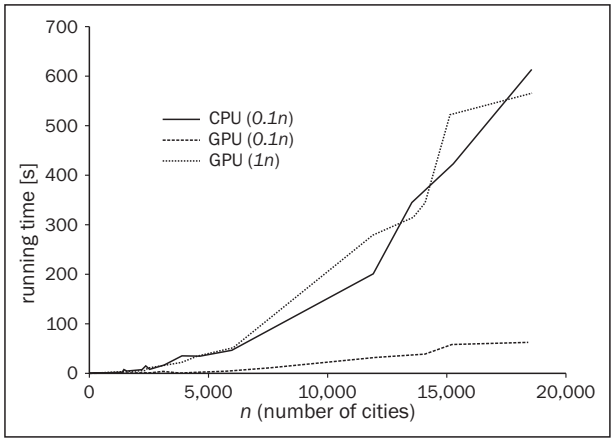

Figure 6 – Comparison of ILS running time on CPU and GPU

of GPU primarily because of the speed-up that can reach up to 20 times in our implementation of 2-opt and 3-opt local searches with the respect to CPU (*Figure 5* and *Figure 6*). In *Figure 6* it can be seen that ILS *1n* on the GPU performs within a similar time as CPU ILS *0.1n* but with significantly lower deviation (*Table 3*, Δ_{CPU} is 2.0 and Δ_{GPU} is 0.78) from the optimal solution.

The best achievement we have obtained in this research is that we can find solutions that are on the average 0.3% worse than the optimal solutions for all tested problems. Furthermore, the test showed that our GPU implementation of ILS algorithm is up to 20 times faster than the CPU implementation of ILS (*Table 3*). Because of this 10 times more ILS iterations can be performed and higher quality solutions achieved.

5.1 CRO TSP benchmark

CRO TSP benchmark problem (*Figure 7*) is constructed from 6,857 urban and rural locations of post offices in Croatia. The unusual shape of the Croatian territory boundary and high density of population in the North of the country are specific and could be an interesting property of the proposed CRO TSP problem.

Figure 7 - CRO TSP benchmark

For the CRO TSP problem the ILS *10n* algorithm gives the total Euclidian distance of 157,209 on GPU and 157,247 on CPU. The reason for the difference between the results for CPU and GPU is the same as was commented for *Table 3*. The average execution time on GPU is 632 *sec* and on CPU is 8,770 *sec* that gives 14 times speed-up on the GPU. One can conclude that the practical problem is in line with the standard TSPlib problem results (see *Table 1*, group 2000-18512). Furthermore, one can see that the practical problems, which are usually large problems, can greatly benefit from the ILS GPU implementation. More details on the Euclidian CRO TSP instance could be found in [26].

6. CONCLUSION

The main challenges in using GPU cards for optimization problems are efficient data flow optimization, effective communication among threads in various blocks, and branching. The branching problem arises particularly when more advanced local search algorithms are used.

The main reason to start working on the GPU implementations of the 2-opt and 3-opt local search is to investigate the possibility to deploy local search operators on data-flow multiprocessor architecture such as GPU. The main problem of the big instances of the TSP is that NP-hard problems, even the heuristic algorithm, require long execution time.

In the time of writing this paper, there were just a few papers that investigated the possibilities of using GPU graphic cards in solving TSP problem using heuristics. The major achievement of this paper is the implementation of the 2-opt and 3-opt local search on GPU. We showed that our implementation can greatly outperform the CPU implementation in the execution time (up to 27 times for problems larger than ~85,000 cities). The advanced local search algorithm can also benefit from our 2-opt and 3-opt local search GPU implementation. Our GPU Iterative Local Search algorithm can find solutions that are on the average 0.3% far from the optimal solutions. Furthermore, it has also been shown that the significant speed-up can be achieved. For the TSPlib problems the maximum speed-up is more than 20 times and for the CRO TSP problem with 6,857 cities, the speed-up is 14 times.

There are still many open ideas for further research. Some of the most interesting topics that will definitely benefit from the GPU are efficient implementation of DLB and the first improvement approach, which could significantly decrease the overall execution time. Further research should also include experimentations with the different initial solutions as well as with different neighbours list that will make our implementation more competitive in the solution quality to the state-of-the-art algorithms.

JURAJ FOSIN, doktorand
E-mail: juraj.fosin@fpz.hr
Sveučilište u Zagrebu, Fakultet prometnih znanosti
Vukelićeva 4, 10000 Zagreb, Hrvatska
DAVOR DAVIDOVIĆ, doktorand
E-mail: davor.davidovic@irb.hr
Ruđer Bošković Institute
Bijenička cesta 54, HR-10000, Zagreb, Hrvatska
Dr. sc. **TONČI CARIĆ**
E-mail: tonci.caric@fpz.hr
Sveučilište u Zagrebu, Fakultet prometnih znanosti
Vukelićeva 4, 10000 Zagreb, Hrvatska

SAŽETAK

GPU IMPLEMENTACIJA OPERATORA LOKALNOG PRETRAŽIVANJA ZA SIMETRIČAN PROBLEM TRGOVAČKOG PUTNIKA

Problem trgovačkog putnika (TSP) je često proučavan problem kombinatorne optimizacije i bitan je za mnoge praktične primjene u području transporta. TSP je NP-težak problem, za čije je optimalno rješavanje egzaktnim algoritmima potrebna značajna računalna snaga. Zadnjih godina brz razvoj grafičkih procesnih jedinica (GPU) za opću namjenu donio je mogućnost značajnog smanjenje vremena izvršavanja algoritama. U ovom radu implementirali smo 2-opt i 3-opt operatore lokalnog pretraživanja za rješavanje problema trgovačkog putnika na GPU koristeći CUDA sučelje za programiranje. Doprinos ovog rada očituje se u paralelnoj implementaciji iterativnog lokalnog pretraživanja s 2-opt i 3-opt operatorima za simetričan problem trgovačkog putnika koji je optimiziran za izvršavanja na GPU-u. Opisani algoritam rješava velike probleme trgovačkog putnika (do 85,900 gradova). U radu je pokazano da GPU implementacija može biti i do 27 puta brža od implementacije na centralnoj procesnoj jedinici (CPU) bez da se izgubi kvaliteta rješenja. Rezultati su dani za probleme iz TSPlib biblioteke kao i za predloženi CRO TSP problem.

KLJUČNE RIJEČI

problem trgovačkog putnika, operatori lokalnog pretraživanja, 3-opt, paralelno iterativno lokalno pretraživanje, grafička procesna jedinica, CUDA

REFERENCES

[1] **Garey, M.R.**, **Johnson, D.S.**: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979
[2] http://graphics.stanford.edu/projects/brookgpu/, last accessed on June 2012
[3] http://www.nvidia.com/, last accessed on June 2012
[4] http://www.amd.com/us/products/technologies/stream-technology/pages/stream-technology.aspx, last accessed on June 2012
[5] **Johnson, D.S.**, **McGeoch, L.A.**: *The traveling salesman problem: a case study in local optimization*, Local Search in Combinatorial Optimization, 1997
[6] **Mertz, P.**, **Freisleben, B.**: *Memetic algorithms for the traveling salesman problem*, Complex Systems, 2001
[7] **Johnson, D.S.**, **McGeoch, L.A.**: *Experimental Analysis of Heuristics for the STSP, The Traveling Salesman Problem and Its Variations*, volume 12, Springer US, 2004
[8] **Hoos, H.H.**, **Stützle, T.**: *Stochastic Local Search*, Morgan Kaufman, 2005
[9] **Bentley, J.J.**: *Fast algorithms for geometric traveling salesman problems*, ORSA Journal on Computing, 1992
[10] **Croes, G.A.**: *A method for solving traveling-salesman problems*, Operations Research, 1958
[11] **Lin, S.**: *Computer solutions of the traveling salesman problem*, Bell System Tech, 1965
[12] **Lin, S.**, **Kernighan, B.W.**: *An effective heuristic algorithm for the traveling salesman problem*, Operations Research, 1973
[13] **Helsgaun, K.**: *General k-opt submoves for the Lin–Kernighan TSP heuristic*, Mathematical Programming Society, 2009
[14] **Zhao, J.**, **Liu, Q.**, **Wang, W.**, **Wei, Z.**, **Shi, P.**: *A parallel immune algorithm for traveling salesman problem and its application on cold rolling scheduling*, Information Sciences, 2011
[15] **Luong, T.V.**, **Melab, N.**, **Talbi, E.G.**: *GPU-based island model for evolutionary algorithms*, Proceedings of the 12th annual conference on Genetic and evolutionary computationGECCO '10, ACM, 2010
[16] **Dorigo, M.**, **Maniezzo, V.**, **Colorni, A.**: *The ant system: Optimization by a colony of cooperating agents*, IEEETransactions on Systems, Man, and Cybernetics-Part B, vol. 26, pp.2941, 1996
[17] **Bai, H.**, **Yang, D.**, **Li, X.**, **He, L.**, **Yu, H.**: *Max-min ant system on GPU with CUDA*, Fourth International Conference on Innovative Computing, Information and Control, 2009
[18] **You, Y.**: *Parallel ant system for traveling salesman problem on GPUs*, Eleventh Annual Conference on Genetic and Evolutionary Computation, 2009
[19] **Lin, Y.**, **Cai, H.**, **Xiao, J.**, **Zhang, J.**: *Pseudo parallel ant colony optimization for continuous functions*, International Conference on Natural Computation, 2007
[20] **Cecilia, J.**, **Garcia, J.**, **Nisbet, A.**, **Amos, M.**, **Ujaldon, M.**: *Enhancing data parallelism for ant colony optimisation on GPUs*, Journal of Parallel and Distributed Computing ,2012
[21] **O'Neil, M.A.**, **Tamir, D.**, **Burtscher, M.**: *A parallel gpu version of the traveling salesman problem*, International Conference on Parallel and Distributed Processing Techniques and Applications, 2011
[22] **Luong, T.V.**, **Melab, N.**, **Talbi, E.G.**: *Parallel local search on GPU*, Report 6915, INRIA, 2009
[23] **Helsgaun, K.**: *An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic*, European Journal of Operational Research, 2000
[24] *CUDA C Programming Guide*, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, last accessed on June 2012
[25] **Reinelt, G.**: *TSPLIB – A traveling salesman problem library*, ORSA JOURNAL ON COMPUTING 3 (1991) 376–384
[26] *URL for Euclidian CRO TSP*, http://fulir.irb.hr/504/1/CRO6857.tsp