

We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

6,900

Open access books available

186,000

International authors and editors

200M

Downloads

Our authors are among the

154

Countries delivered to

TOP 1%

most cited scientists

12.2%

Contributors from top 500 universities



WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index
in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com



CUDA Accelerated 2-OPT Local Search for the Traveling Salesman Problem

*Donald Davendra, Magdalena Metlicka
and Magdalena Bialic-Davendra*

Abstract

This research involves the development of a compute unified device architecture (CUDA) accelerated 2-opt local search algorithm for the traveling salesman problem (TSP). As one of the fundamental mathematical approaches to solving the TSP problem, the time complexity has generally reduced its efficiency, especially for large problem instances. Graphic processing unit (GPU) programming, especially CUDA has become more mainstream in high-performance computing (HPC) approaches and has made many intractable problems at least reasonably solvable in acceptable time. This chapter describes two CUDA accelerated 2-opt algorithms developed to solve the asymmetric TSP problem. Three separate hardware configurations were used to test the developed algorithms, and the results validate that the execution time decreased significantly, especially for the large problem instances when deployed on the GPU.

Keywords: traveling salesman problem, CUDA, 2-opt, local search, GPU programming

1. Introduction

This research addresses two very important aspects of computational intelligence, algorithm design, and high-performance computing. One of the fundamental problems in this field is the TSP, which has been used as a poster child for the notorious $\mathcal{P} = \mathcal{NP}$ assertion in theoretical computer science.

TSP in nominal form is considered *NP-Complete*, when attempted using exact deterministic heuristics. The time complexity when solving it using the *Held-Karp algorithm* is $O(n^2 2^n)$ and the space complexity is $O(n 2^n)$. When solving the problem using optimization algorithms and approximation, then problem tends to be *NP-Hard*.

2-opt is considered the simplest local search for the TSP problem. Theoretical knowledge about this heuristic is still very limited [1]; however, simple euclidean distance variants have been shown to have complexity of $O(n^3)$ [2]. Generally, the computed solution has been shown to be within a few percentage points of the global optimal [3].

One of the empirical approaches of improving the execution of the algorithm is applying high performance computing (HPC) paradigm to the problem. This is generally possible if the problem is deducible to a parallel form.

A number of different HPC approaches exist, namely, *threads*, *OpenMP*, *MPI* and *CUDA*. *CUDA* is by far the most complex and accelerated approach, as it requires programming on the GPU instead of the central processing unit (CPU).

Since its inception, *CUDA* has been widely used to solve a large number of computational problems [4]. This research looks to harness this approach to implement the 2-opt approach to the TSP problem.

The outline of the chapter follows with the introduction of the mathematical background of the TSP problem followed by the 2-opt algorithm. *CUDA* is subsequently discussed and the two *CUDA* developed 2-opt algorithm variants are described. The experimentation design discusses the hardware specifications of the three different architectures and then the obtained results are discussed and analyzed in respect to the execution time.

2. Traveling salesman problem

The TSP is a well-studied problem in literature [5, 6], which in essence tries to find the shortest path that visits a set of customers and returns to the first. A number of studies have been done using both approximation-based approaches [7] and metaheuristics. Metaheuristics are generally based on evolutionary approaches. A brief outline of different approaches can be obtained from:

1. Tabu Search: [8]
2. Simulated Annealing: [9]
3. Genetic Algorithm: [10, 11]
4. Ant Colony Optimization: [12]
5. Particle Swarm Optimization: [13]
6. Cuckoo Search: [14]
7. Firefly Algorithm: [15]
8. Water Cycle Algorithm: [16]
9. Differential Evolution Algorithm: [17]
10. Artificial Bee Colony: [18]
11. Self Organizing Migrating Algorithm: [19]

The TSP function can be expressed as shown in Eq. (1).

$$x_{ij} = \begin{cases} 1 & \text{the path goes from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where $x_{ij} = 1$ if city i is connected with city j , and $x_{ij} = 0$ otherwise. For $i = 0, \dots, n$, let u_i be an artificial variable and finally take c_{ij} to be the distance from city i to city j . The objective function can be then formulated as Eq. (2):

$$\begin{aligned}
 & \min \sum_{i=0}^n \sum_{j \neq i, j=0}^n c_{ij} x_{ij} \\
 & 0 \leq x_{ij} \leq 1 \quad i, j = 0, \dots, n \\
 & u_i \in \mathbb{Z} \quad i = 0, \dots, n \\
 & \sum_{i=0, i \neq j}^n x_{ij} = 1 \quad j = 0, \dots, n \\
 & \sum_{j=0, j \neq i}^n x_{ij} = 1 \quad i = 0, \dots, n \\
 & u_i - u_j + nx_{ij} \leq n - 1 \quad 1 \leq i \neq j \leq n
 \end{aligned} \tag{2}$$

3. 2-OPT algorithm

The 2-opt algorithm is one of the most famous heuristics developed originally for solving the TSP problem. It was first proposed by Croes [20]. Along with 3-opt, generalized as k -opt [21], these heuristics are based on exchange of up to k edges in a TSP tour (more information on application of k -opt local search techniques to TSP problems can be obtained from [22]). Together they are called exchange or local improvement heuristics. The exchange is considered to be a single move, from this point of view, such heuristics search the neighborhood of the current solution, that is, perform a local search and provide a locally optimal solution (k -optimal) to the problem [23].

The 2-opt procedure requires a starting feasible solution. It then proceeds by replacing the two non-adjacent edges, (v_i, v_{i+}) and (v_j, v_{j+}) by (v_i, v_j) and (v_{i+}, v_{j+}) , and reversing one of the subpaths produced by dropping of edges, in order to maintain the consistent orientation of the tour. For example, the subpath $(v_i, v_{i+}, \dots, v_j, v_{j+})$ is replaced by $(v_i, v_j, \dots, v_{i+}, v_{j+})$. The solution cost change produced in this way can be expressed as $\Delta_{ij} = c(v_i, v_j) + c(v_{i+}, v_{j+}) - c(v_i, v_{i+}) - c(v_j, v_{j+})$. If $\Delta_{ij} < 0$, the solution produced by the move improves upon its predecessor. The procedure iterates until no move where $\Delta_{ij} < 0$ (no improving move) can be found [24].

The 2-opt local search was described by Kim et al. [25] as follows:

Step 1: Let S be the initial solution, $f(S)$ its objective function value. Set $S^* = S, i = 1, j = i + 1 = 2$.

Step 2: Consider exchange result S' such that $f(S') < f(S^*)$. Set $S^* = S'$. if $j < n$ repeat step 2. Otherwise set $i = i + 1$ and $j = i + 1$. if $i < n$ repeat step 2, otherwise go to step 3.

Step 3: if $S \neq S^*$ set $S = S^*, i = 1, j = i + 1$ and go to step 2. Otherwise output best solution S and terminate the process.

4. CUDA

General purpose GPU computing (GPGPU) programming was introduced by Apple Cooperation, which created the Kronos Group [26] to further develop and promote this new approach to accelerate scientific computing paradigms.

GPU's offer significantly faster acceleration due to their unique hardware architecture. GPGPU's started to increase in application from 2006. At this point NVIDIA decided to create its propriety unique architecture called Compute Unified Device Architecture (CUDA), specific for their *Tesla* generation GPU cards. In order to support this architect, specific API primitive extensions of C, C++ and Fortran extensions has been developed [27, 28].

The specific C/C++ language extension for the C language is called the CUDA-C. This contains a number of accelerated libraries, extensions, and APIs. These are scalable and freely available without professional license. The main computational bottleneck is the splitting of the task between GPU and CPU tasks, where CPU handles better memory management and memory checking and GPU handles the data acceleration using parallization. It is considered *heterogenous* programming, where compute intensive data parallel tasks are offloaded on to the GPU.

CUDA contains three specific paradigms, *thread hierarchy*, *memory hierarchy* and *synchronization*. These can be further divided into *coarse-grained* parallelism on the *blocks* in *grid* parallization and *fine-grain* parallization in the *threads* in *block*, which requires low-level synchronization.

4.1 Thread hierarchy

CUDA kernels are special function calls, which is used for data parallization. Each kernel launches *threads* which are grouped into *blocks* which are then grouped into *grids*. Communication is done synchronously by *threads* in a *block*, whereas *blocks* are independent. Certain programming techniques needs to be undertaken to ensure data synchronization and validity between *blocks*. *Threads* in different *blocks* are not able to communicate with each other.

Threads are distinguished by their unique *threadId* in their respective *blockId*, which allows operating on specific data in the *global* and *shared* memory.

4.2 Memory hierarchy

There are different memory types in the GPU, which CUDA can utilize. Some memory structures are based on *cache*, some are read-only, etc. The first higher level memory structure is called the *global memory*, which can be accessed by all memory *blocks*. Due to its size and access level, it is the slowest memory on the GPU. The second memory level is the *shared memory*, which is shared by blocks, which threads within blocks can access. The third memory is the register memory, which are only accessible by threads, and can be used to local variables. This is the smallest and fastest memory in the GPU. If there are larger memory structures, and when registers are not sufficient, *local* memory can be then utilized. Another memory is *constant* memory which cannot be changed by the kernel code. The final memory is the *texture* memory, which is a read-only cache that provides a speed-up for locality in data access by threads [29].

4.3 Synchronization

Blocks in *grids* are used in coarse-grained parallelism and *threads* in a specific *block* are used in *fine-grained* parallelism. Data sharing in the scope of a kernel is done by *threads* in the *block*. The number of *threads* are limited by the device architecture design (max. 1024) and also by *thread* memory resource consumption. There is a level of scalability as the *blocks* are scheduled independently. Each *block* is assigned to a streaming multiprocessor (MS) in the GPU [29, 30].

5. CUDA-based 2-opt algorithm

This section presents the parallel CUDA-based version of 2-opt algorithm. This is a modification of the local search for permutative flowshop with makespan criterion problem [31] and its NEH variant [32]. Before coming to the parallel implementation description, however, the more detailed pseudocode of sequential version is provided in Algorithm 5, in order to enable better understanding of the CUDA algorithm design.

Algorithm 1: 2-opt sequential version. The $Swap(T, j, i)$ procedure swaps j -th and i -th cities of tour T

Input:
 S : initial solution

```

1      // number of cities
2   $N \leftarrow \text{Size}(S)$ 
      // objective function value of  $S$ 
3   $f_S \leftarrow f(S)$ 
      // temporary solution memory
4   $T \leftarrow S$ 
5  while ImprovementFound do
6      ImprovementFound  $\leftarrow$  False
7      for  $i=1$  to  $N-1$  do
8          for  $j=i+1$  to  $N$  do
9               $T \leftarrow \text{Swap}(T, i, j)$ 
10              $f_T \leftarrow f(T)$ 
11             if  $f_T < f_S$  then
12                  $S \leftarrow T$ 
13                  $f_S \leftarrow f_T$ 
14                 ImprovementFound  $\leftarrow$  True
15                 break(2)
16             end
17              $T \leftarrow \text{Swap}(T, j, i)$ 
18         end
19     end
20 end
21 return  $S$ 
    
```

As can be seen already from the analysis of description of 2-opt, the task that can be done in parallel is the exploration of neighborhood of the current solution. This is divided between individual CUDA blocks. Possible neighbors of the current solution are split evenly between the launched *blocks*, which then explores these

neighbors evenly including the fitness evaluations. If a new better solution is found, it is then stored into the *global memory* allocation of that block. Thereafter, if at least one of the launched *blocks* finds an improving solution during the iteration, the best cost solution amongst all *blocks* is obtained and stored into memory as the current solution for the next iteration. Otherwise, the current solution is returned as the best. It should be noted that the fitness function is not parallelized, as only a single *thread* in each *block* is tasked with this task.

Each *block* explores approximately the same amount of possible neighbors to the current solution (in the worst case, when no improving solution is found), including the cost evaluation. However, if it finds an improving solution, that solution is stored into the *global memory* allocated for each *block*, and the *block* terminates. If at least one of the *blocks* found an improving solution, the minimal cost solution amongst all *blocks* is found and stored into memory as the current solution for the next iteration. Otherwise, the current solution is returned. The cost function evaluation itself was not parallelized, as in each *block* only a single *thread* performs this task.

The outline of the parallel algorithm can be given as follows:

Step 1: Set current solution S = Initial solution.

Step 2: Explore the neighborhood of S by G blocks in parallel. In each block b :

Step 1.1: Determine initial index i for b .

Step 1.2: Explore all neighbors of S created by swapping of i and $j, j \in \{1, \dots, N\}$. If improving neighbor T found, go to *step 1.4*.

Step 1.3: Determine next index i for b . If $i \geq N$, terminate. Otherwise go to *step 1.2*.

Step 1.4: Store T and its objective function value f_T into global memory and terminate.

Step 3: If no improving solution found, exit procedure and return S as the best solution found. Otherwise determine the best solution amongst those found by blocks in parallel.

Step 4: Store best solution as S . Go to *step 2*.

Where N is the number of cities in the tour and i is the outer loop index (see Algorithm 1 for sequential version of 2-opt).

5.1 Exploration and evaluation of neighboring solutions

The neighbors of solution are generated and evaluated in this kernel. From the sequential version pseudocode (Algorithm 1), it is obvious that the function of generating individual neighbors by swapping every possible pair of jobs pair-wise (i, j) for $i = 1, \dots, N$ and $j = i + 1, \dots, N$ can be considered independent and therefore executed in parallel. These solutions can be stored in the *shared memory* after generation. After evaluation, if the new solution has better fitness value compared to the current one, it is stored into the *global memory* allocated for each *block*, to avoid data races between *blocks* (this is illustrated in **Figure 1** depicting memory layout for six cities and four blocks). The improvements counter in the *global memory* is incremented using an atomic operation. This counter is compared against zero after the kernel termination, to determine if the stopping criterion of the algorithm was met. The fitness function itself is evaluated by only a single *thread*; the other *threads* in a *block* process the elements of the solution when transferring data between *shared* and *global memory* locations.

It is logically impractical to allocate the full number of $(N - 1)^2 / 2$ *blocks* on the GPU in most case scenarios. This number can be very large, whereas the number of SMs and the number of resident *blocks* on SM is limited by various factors, such as the number of *threads* in a *block* and a *registers/shared memory* usage.

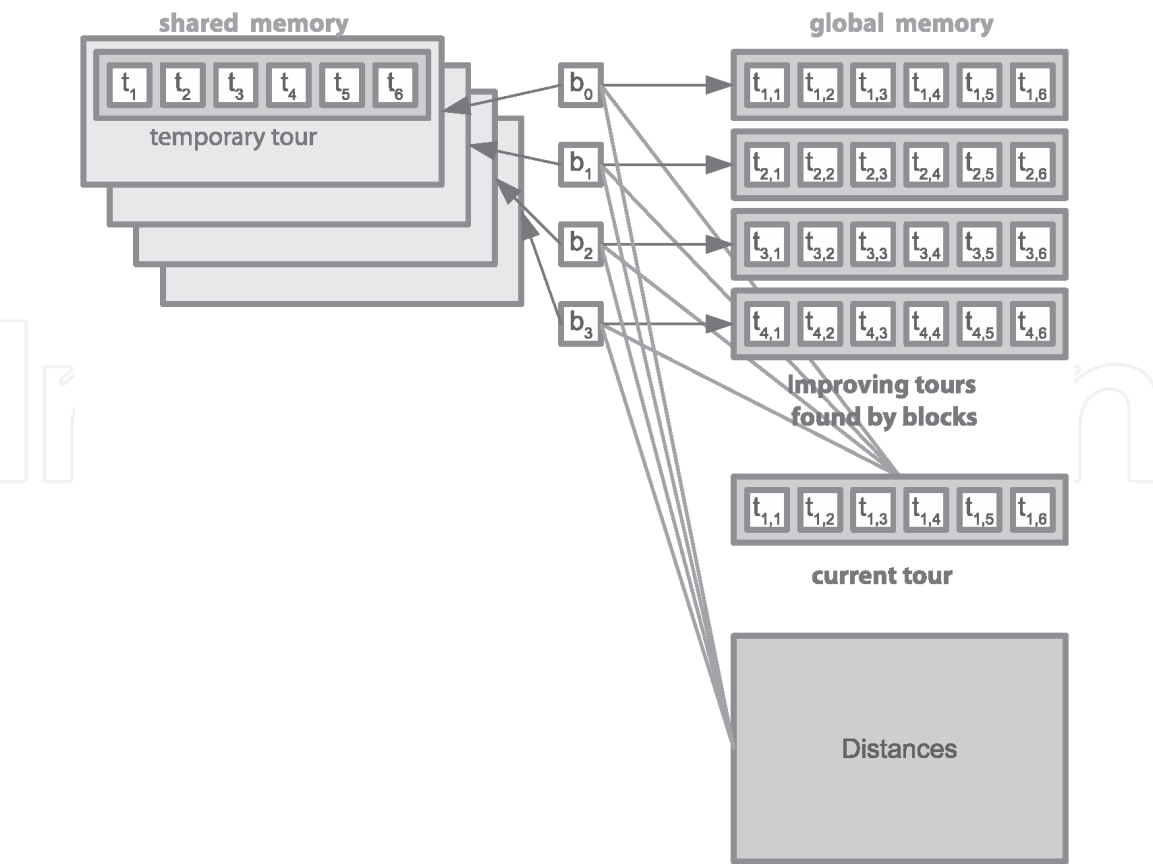


Figure 1.
CUDA-based 2-opt memory layout.

The optimal number of *threads* in a *block* maximizing the number of resident *blocks*, as well as GPU occupancy, can be easily determined based on the calculations performed in the CUDA occupancy calculator tool [33], as a function of the number of cities in a tour (which determines the size of *shared memory* used). This can maximize the utilization of the GPU, while reducing the total *global memory* size required by the *grid*, as well as the workload done by the search for minimal cost solution in the next kernel. The mapping of the *blocks* to the tasks however can become more complicated to implement in code.

Using the assumption that the number of *blocks* will be nearly always smaller than the aforementioned function of the number of actual cities for the problem instances of interest (problems with cities larger than 30), only the outer loop of the sequential 2-opt algorithm was parallelized. The inner loop is performed by each *block* sequentially. This reduces the data transfers between *global* and *shared memory*, and does not eliminate the advantage of the low complexity of the swap operation at the same time. If the solution created by swapping jobs i and j is worse than the current one, it is easy to reverse this change by swapping again j and i , with equal complexity. Therefore, maximally $N - 1$ *blocks* are needed for this function. The mapping of *blocks* to tasks is illustrated in **Figure 2**.

5.2 Parallel reduction to obtain minimal cost

The *parallel reduction* procedure is used to find the index of the solution with the minimal fitness value. This employs *shared memory* to store the data being used, whereas the data is initially copied from the *global* to *shared memory*. In this step, each active thread compares two costs, and stores the smaller of the two costs on the place of the first cost, along with its original index (cost is represented as a structure

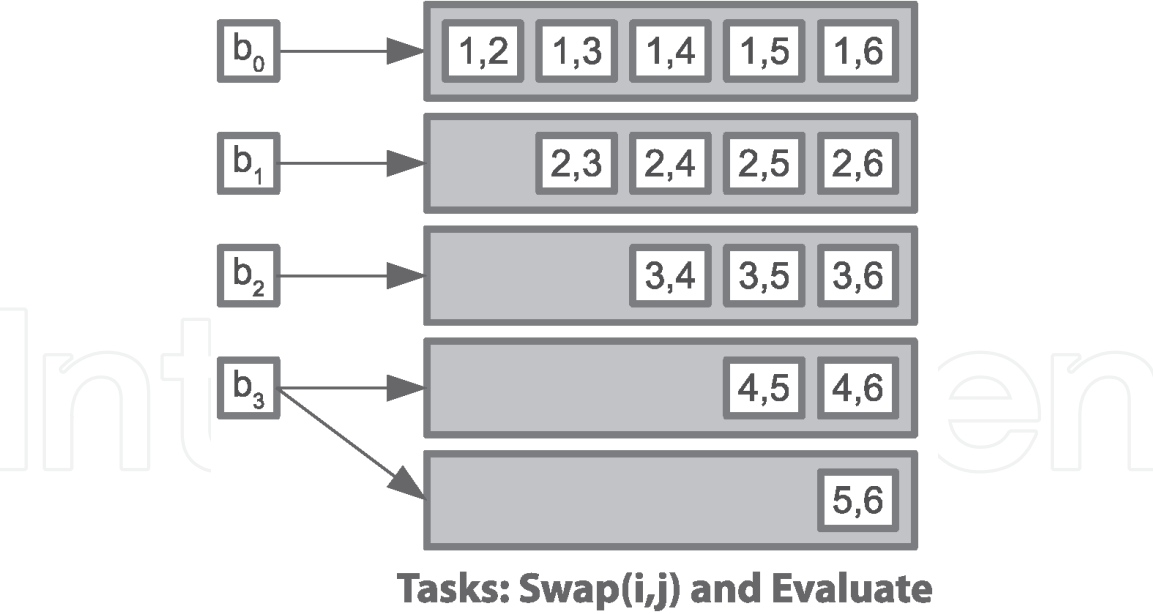


Figure 2.
CUDA-based 2-opt, mapping of blocks to tasks.

containing two elements: cost value, and cost index). Using this reduction, the first element of the costs array contains the minimal cost found, along with its respective solution index. This pair is then written into *global memory*.

5.3 Device synchronization and subsequence update

In the final process, a new kernel copies the best indexed solution into the current solution buffer, and the next step of the main loop can be performed. A global CUDA device synchronization is required for relatively large data (for a tour size/number of *threads* in a *block* of size more than approximately 100, as was empirically confirmed) before the start of the synchronization. As each of the

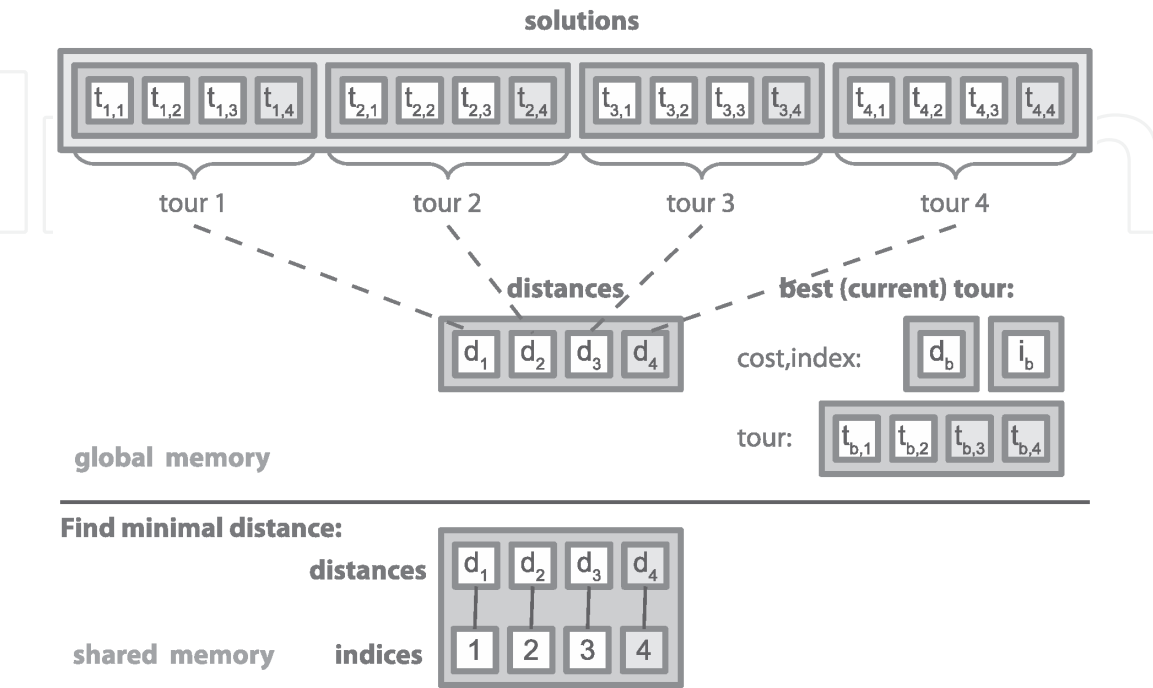


Figure 3.
CUDA-based 2-opt distance and indices layout.

kernels consumes some of the GPU resources, it is necessary to wait, until the pending kernels completely finish the execution, and release their resources, otherwise the GPU freezes and unsuccessful kernel launches start to appear. This is done by calling `cudaDeviceSynchronize()` function from the host code, after the Update kernel is launched.

Figure 3 outlines the memory layout of the previously described code (without TSP input data, for the current subsequence size 2, city tour 4. The data fields not used in the current step are grayed out). The candidate solutions are stored in one *global memory* 1D array, which conceptually represents 2D array, wherein each row contains one candidate tour. The respective costs are stored in a separate array. The TSP problem input data (distance between cities) are stored in the similar fashion in *global memory* (because of its large size).

This implementation is expected to provide in each step the speedup proportional to the number of solutions generated.

6. 2-OPT variants

Two versions of the 2-opt local search was implemented in this work. The first is the **LS2OPT** variant, which uses the search with the *first ascend strategy*. In this strategy, the next tour is the *first* improving solution found. This can be given in Algorithm 2.

The second variant is the **MLS2OPT** version, which is the *best ascend strategy*. In this strategy, the next tour is the *best* improving solution found in the 2-swap neighborhood as given in Algorithm 3.

7. Experimentation design

The experimentation design is as the following. Three different CPU's and three different GPU's are used to run the two different 2-opt variants on a selected number of asymmetric TSP instances (ATSP). The only measure is the time complexity.

The problem instances of the ATSP was obtained from the TSP library [34]. The following problems were selected due to differing city sizes as given in **Table 1**.

The machine specifications is given in **Table 2**. Three separate machines were used with differing CPUs and GPUs. Two machines were on a Windows 10 operating system and the other is a Central Washington University Supercomputer cluster running Ubuntu [35]. Machine 2 and 3 utilized headless GPU's.

Data	Cities
ft70	70
ftv64	65
ftv170	171
kro124p	100
rbg323	323
rbg358	358
rbg403	403

Table 1.
TSP instances and number of cities.

Specifications	Machine 1		Machine 2		Machine 3	
Processor	Intel i7-9750H	GTX 1050	Intel i7-7800X	Titan Xp	Power 8	P100
Memory	16 GB	2 GB	32 GB	12 GB	32 GB	16 GB
Cores	4	640	6	3840	6	3584
OS	Win10		Win10		Ubuntu	
Language	C++	CUDA-C	C++	CUDA-C	C++	CUDA-C
IDE	Visual Studio 17		Visual Studio 17		Makefile	
Cost (USD)	\$200		\$1500		\$15,000	

Table 2.
Machines specifications.

Algorithm 2: LS2OPT sequential version. The $Swap(T,j,i)$ procedure swaps j -th and i -th cities of tour T

```
1 for i=1 to N-1 do
2   for j=i+1 to N do
3      $T \leftarrow Swap(T,i,j)$ 
4      $f_T \leftarrow f(T)$ 
5     if  $f_T < f_S$  then
6        $S \leftarrow T$ 
7        $f_S \leftarrow f_T$ 
8        $ImprovementFound \leftarrow True$ 
9       break
10    end
11     $T \leftarrow Swap(T,j,i)$ 
12  end
13 end
14 return S
```

Algorithm 3: MLS2OPT sequential version. The $Swap(T,idx,i)$ procedure swaps idx -th and i -th cities of tour T , where idx -th is the best 2-swap schedule j -th index found after iteration

```
1 for i=1 to N-1 do
2   for j=i+1 to N do
3      $T \leftarrow Swap(T,i,j)$ 
4      $f_T \leftarrow f(T)$ 
5     if  $f_T < f_S$  then
6        $f_{S_j} \leftarrow f_T$ 
7        $ImprovementFound \leftarrow True$ 
8     end
9   end
10  if  $ImprovementFound$  then
11     $idx \leftarrow \min(f_{S_j})$ 
12  end
13   $T \leftarrow Swap(T,idx,i)$ 
14 end
15 return S
```

8. Results and analysis

The results are grouped by the machine architectures, as there is a dependency between the CPU and GPU. Thirty experimentations was done of each problem instance on each machine for each algorithm and the average time is given in the tables (* in msec). The *percentage relative difference* (PRD) is calculated between the CPU and GPU times as given in Eq. (3). Negatives values (given as bolded text in the tables) indicate that the GPU execution is faster.

$$PRD = ((GPU - CPU)/CPU) \cdot 100 \tag{3}$$

The first part of the first machine experiment results of the LS2OPT and its CUDA variant is given in **Table 3**. The first column is the problem instances and the second and third column is the CPU and GPU average results of the LS2OPT in milliseconds. The final column is the PRD results. From all the results, apart from the *ftv64* instance, the GPU produced faster results. The *average* time was **22480.28** ms for the CPU and **2168.57** ms for the GPU. The average PRD was **−47.29%** for all experiments. A deeper analysis shows that for the larger instances, the PRD was over 80%.

The plot of the execution time is given in **Figure 4** where the execution speedup is clearly identifiable for the larger instances.

The second part of the first machine experimentation is the MLS2OPT and its CUDA variant and the result are given in **Table 4**. For all the problem instances, the execution time for the GPU was significantly better. The *average* time was **14183.85** ms for the CPU and **1854.28** ms for the GPU. The average PRD was **−52.55%** for all experiments. Apart from two instances, all the other were above 85% PRD.

The plot of the execution time is given in **Figure 5**, where the execution speedup is linearly identifiable for the larger instances.

The first part of the second machine experiment results of the LS2OPT and its CUDA variant is given in **Table 5**. As the NVidia Titan Xp is a dedicated headless TESLA category GPU, the computational times are better than the CPU for all the results. The *average* time was **12157.14** ms for the CPU and **857** ms for the GPU. The average PRD was **−64.92%** for all experiments. A deeper analysis shows that for the larger instances, the PRD was over 90%. As the transfer overhead for the *PCIe* bus is

Data	Intel i7-9750H LS2OPT	Nvidia GTX 1050 LS2OPTCUDA	PRD (%)
ft70	42	34	−19.047
ftv64	14	30	114.29
ftv170	322	87	−72.98
kro124p	580	111	−80.86
rbg323	43,854	2963	−93.24
rbg358	51,069	4096	−91.98
rbg403	61,481	7859	−87.22
Average	22480.28	2168.57	−47.29

*All results are in milliseconds (ms).

Table 3.
Results of the experiments of *Intel i7-9750H* and *NVidia GTX 1050* on the *LS2OPT* and *LS2OPTCUDA* algorithms.

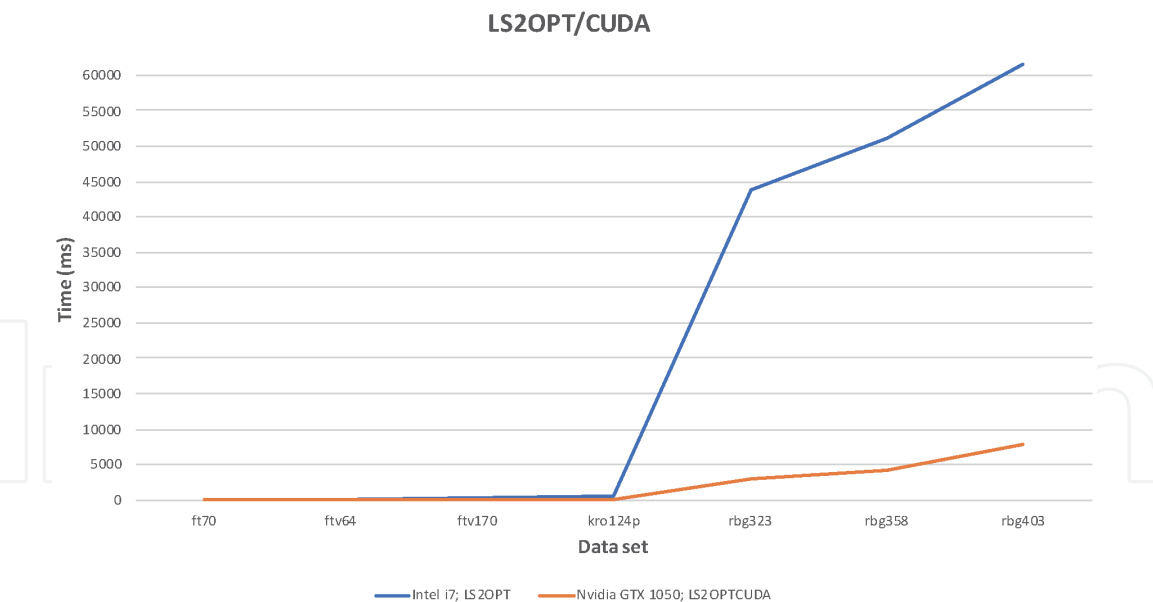


Figure 4. Figure for the experiments of *Intel i7* and *NVidia GTX 1050* on the *LS2OPT* and *LS2OPTCUDA* algorithms.

Data	Intel i7-9750H MLS2OPT	Nvidia GTX 1050 MLS2OPTCUDA	PRD (%)
ft70	37	21	−43.24
ftv64	26	52	100.00
ftv170	619	78	−87.40
kro124p	303	75	−75.25
rbg323	21,205	2525	−88.09
rbg358	31,330	3775	−87.95
rbg403	45,767	6454	−85.90
Average	14183.85	1854.28	−52.55

**All results are in milliseconds (ms).*

Table 4. Results of the experiments of *Intel i7-9750H* and *NVidia GTX 1050* on the *MLS2OPT* and *MLS2OPTCUDA* algorithms.

compensated by more extensive experimentation, larger instances performed faster on the GPU.

The plot of the execution time is given in **Figure 6**, where the execution speedup is clearly identifiable for the larger instances.

The second part of the second machine experimentation is the *MLS2OPT* and its *CUDA* variant and the result are given in **Table 6**. For all the problem instances, the execution time for the GPU was significantly better. The *average* time was **7955.28 ms** for the CPU and **616.28 ms** for the GPU. The average PRD was **−63.39%** for all experiments. The three larger instances were above 90% PRD.

The plot of the execution time is given in **Figure 7**, where the execution speedup is linearly identifiable for the larger instances.

The first part of the third machine experiment results of the *LS2OPT* and its *CUDA* variant is given in **Table 7**. Generally, the *NVidia P100* is regarded as an industry leading GPU solution for scientific computing. This is coupled with the *IBM Power 8 CPU Architecture*. For all the problem instances the result was

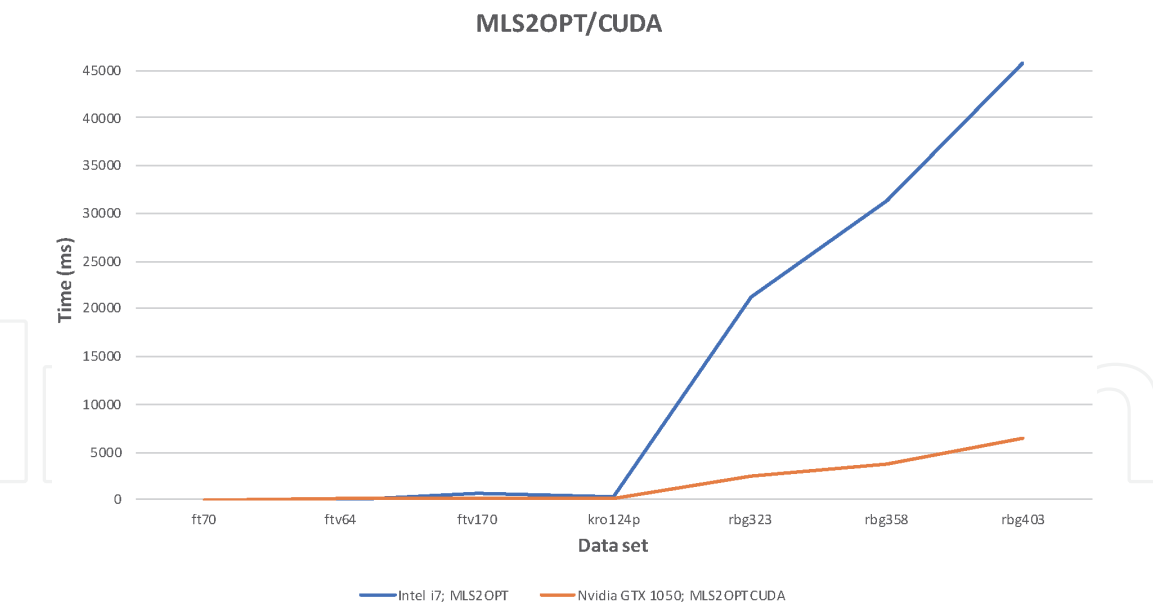


Figure 5. Figure for the experiments of *Intel i7* and *NVidia GTX 1050* on the *MLS2OPT* and *MLS2OPTCUDA* algorithms.

Data	Intel i7-7800X LS2OPT	NVidia Titan Xp LS2OPTCUDA	PRD (%)
ft70	21	18	−14.29
ftv64	12	8	−33.33
ftv170	183	77	−57.92
kro124p	306	94	−69.28
rbg323	23,619	1467	−93.79
rbg358	27,614	1848	−93.31
rbg403	33,345	2487	−92.54
Average	12157.14	857	−64.92

^{*} All results are in milliseconds (ms).

Table 5. Results of the experiments of *Intel i7-7800X* and *NVidia titan Xp* on the *LS2OPT* and *LS2OPTCUDA* algorithms.

significantly better. The *average* time was **28592.43** ms for the CPU and **1536.42** ms for the GPU. The average PRD was **−87.83%** for all experiments.

The plot of the execution time is given in **Figure 8**, where the execution speedup is clearly identifiable for the larger instances.

The second part of the third machine experimentation is the *MLS2OPT* and its CUDA variant and the result are given in **Table 8**. For all the problem instances, the execution time for the GPU was again significantly better. The *average* time was **23429.14** ms for the CPU and **751** ms for the GPU. The average PRD was **−92.78%** for all experiments. The PRD is the highest of all experiments.

The plot of the execution time is given in **Figure 9**, where the execution speedup is linearly identifiable for the larger instances.

The final comparison is of the three GPU’s on the two separate algorithms. **Figure 10** shows the values of the three GPU’s on the problem instances for the *LS2OPTCUDA* algorithm. For the small sized problem, the timing is not significantly distinct. The distinction only becomes variable when the instance sizes increase. Overall, the *NVidia Titan Xp* is the best performing GPU for this algorithm.

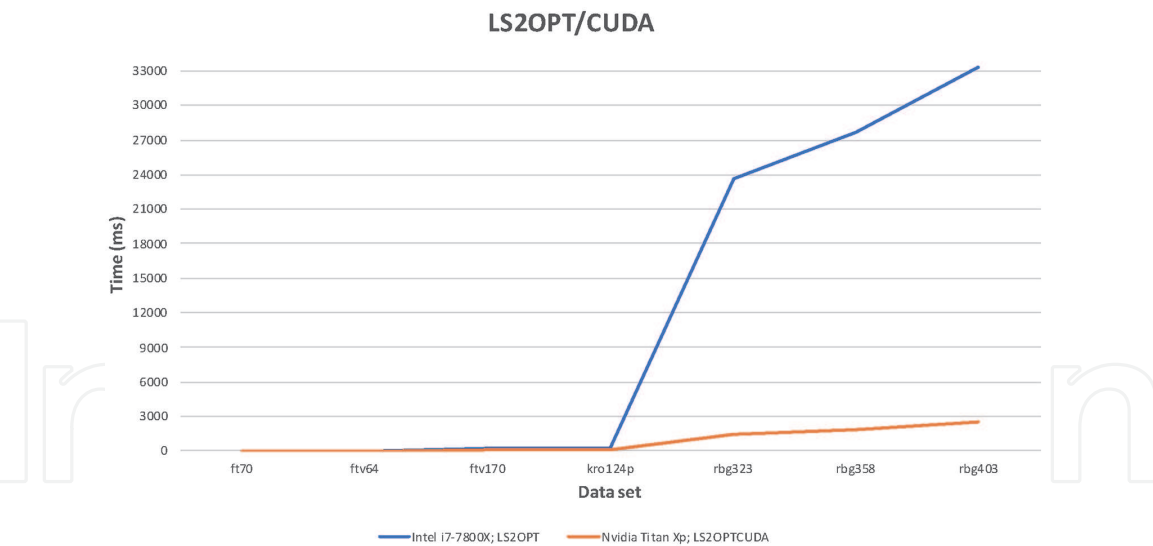


Figure 6. Figure for the experiments of *Intel i7-7800X* and *NVidia titan Xp* on the *LS2OPT* and *LS2OPTCUDA* algorithms.

Data	Intel i7-7800X MLS2OPT	NVidia Titan Xp MLS2OPTCUDA	PRD (%)
ft70	20	16	−20.00
ftv64	11	8	−27.27
ftv170	321	58	−81.93
kro124p	164	56	−65.85
rbg323	11,517	941	−91.83
rbg358	17,109	1059	−93.81
rbg403	24,445	2176	−91.10
Average	7955.28	616.28	−63.39

** All results are in milliseconds (ms).*

Table 6. Results of the experiments of *Intel i7-7800X* and *NVidia titan Xp* on the *MLS2OPT* and *MLS2OPTCUDA* algorithms.

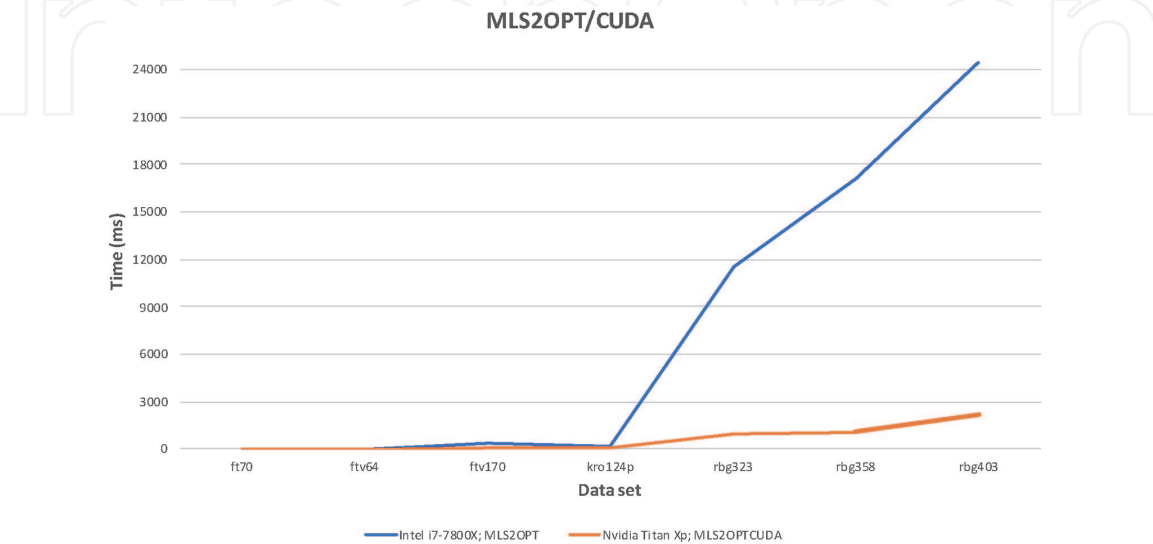


Figure 7. Figure for the experiments of *Intel i7-7800X* and *NVidia titan Xp* on the *MLS2OPT* and *MLS2OPTCUDA* algorithms.

Data	Power 8 LS2OPT	NVidia P100 LS2OPTCUDA	PRD (%)
ft70	57	10	−82.46
ftv64	23	6	−73.91
ftv170	430	75	−82.56
kro124p	754	61	−91.91
rbg323	61,775	3245	−94.75
rbg358	64,419	3587	−94.43
rbg403	72,689	3771	−94.81
Average	28592.43	1536.42	−87.83

** All results are in milliseconds (ms).*

Table 7.
Results of the experiments of **power 8** and **NVidia P100** on the **LS2OPT** and **LS2OPTCUDA** algorithms.

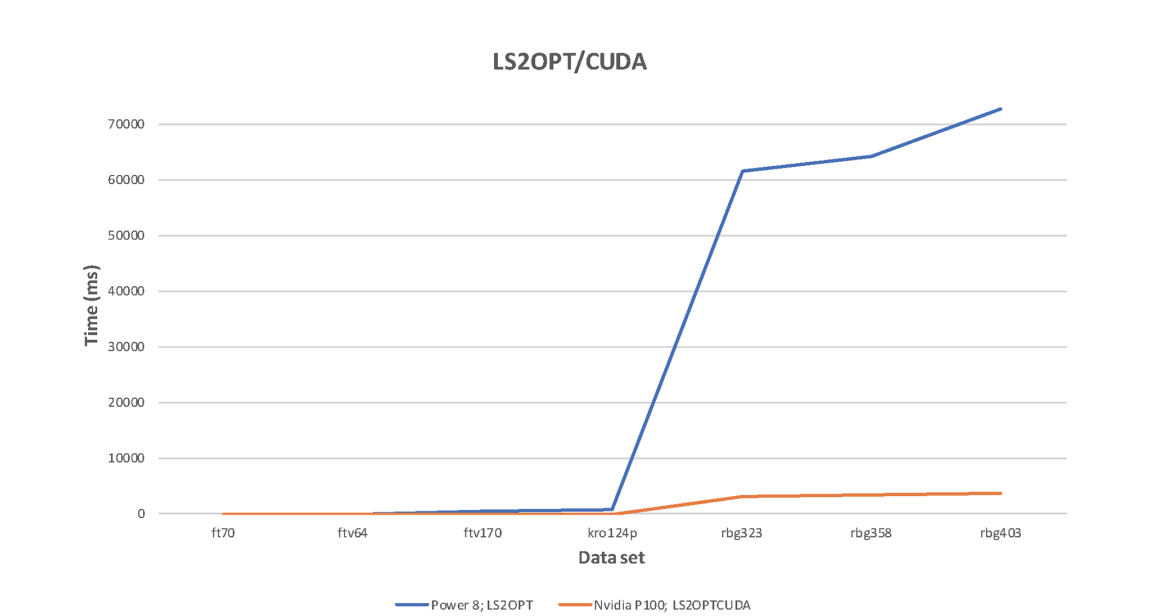


Figure 8.
Figure for the experiments of **power 8** and **NVidia P100** on the **LS2OPT** and **LS2OPTCUDA** algorithms.

Data	Power 8 MLS2OPT	NVidia P100 MLS2OPTCUDA	PRD (%)
ft70	53	7	−86.79
ftv64	33	4	−87.88
ftv170	811	52	−93.59
kro124p	385	35	−90.91
rbg323	30,120	1124	−96.27
rbg358	44,709	1215	−97.28
rbg403	87,893	2820	−96.79
Average	23429.14	751	−92.78

** All results are in milliseconds (ms).*

Table 8.
Results of the experiments of **power 8** and **NVidia P100** on the **MLS2OPT** and **MLS2OPTCUDA** algorithms.

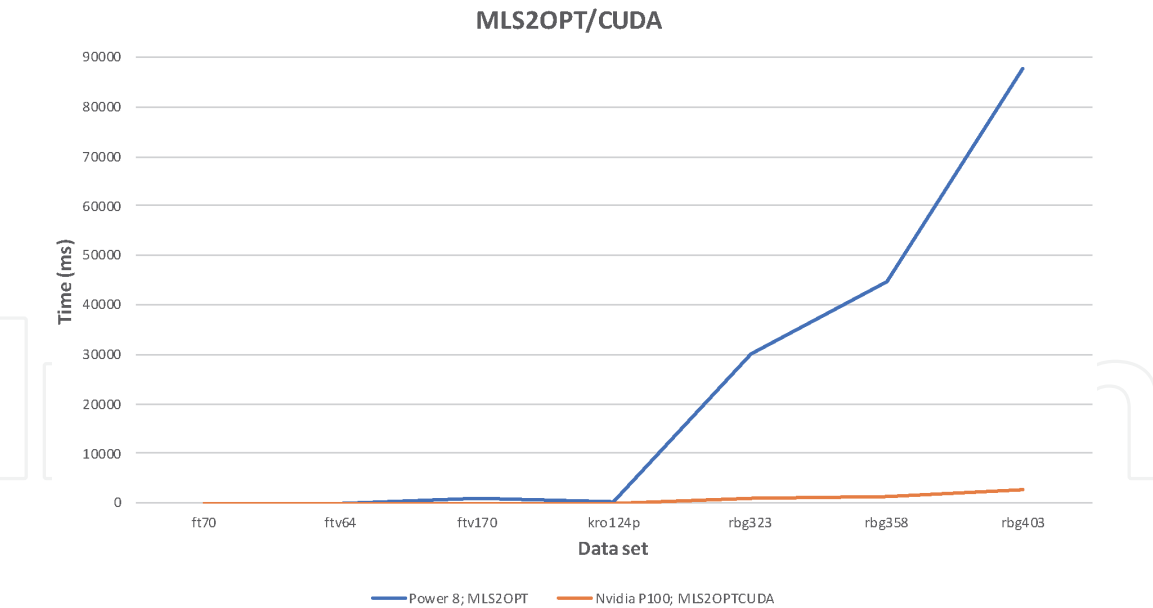


Figure 9.
Figure for the experiments of *power 8* and *NVidia P100* on the *MLS2OPT* and *MLS2OPTCUDA* algorithms.

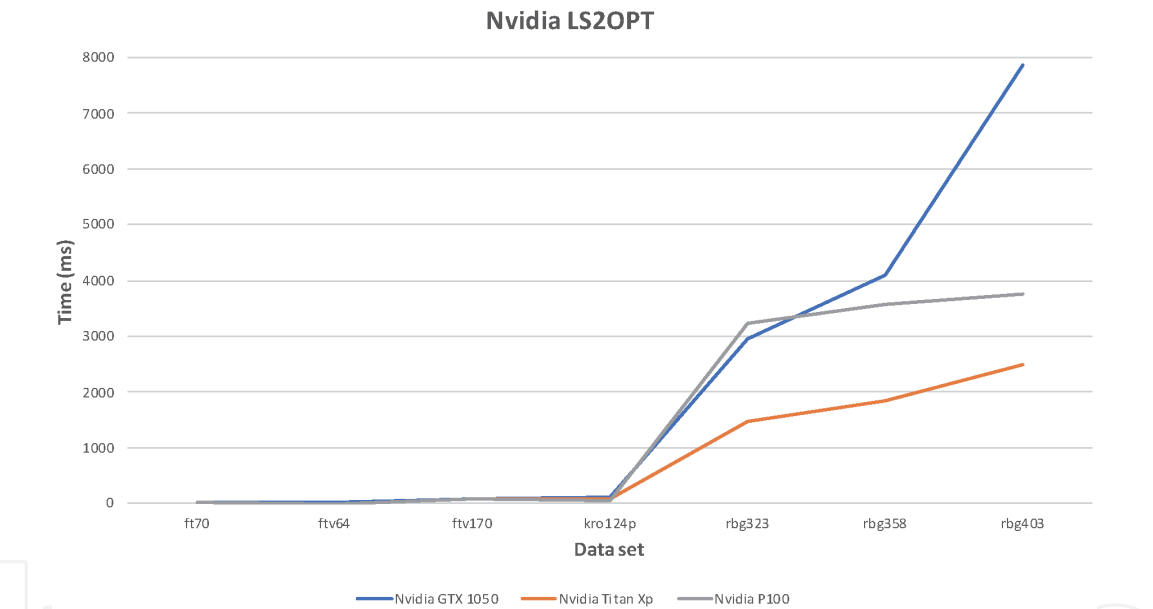


Figure 10.
Figure for the experiments of the three *NVidia GPU's* for the *LS2OPTCUDA* algorithm.

Figure 11 shows the results of the *MLS2OPTCUDA* algorithm on the problem. As with the previous case, the distinction only becomes obvious for large sized problem instances. Again the *NVidia Titan Xp* is the best performing GPU for this algorithm.

9. Algorithm comparison

This section discusses the tour cost obtained by the two different 2-OPT approaches developed here compared with published research. The first comparison is done with the best known solution in literature, which can be obtained from the TSPLib [36].

Table 9 gives the comparison results between the optimal and the results obtained from the *LS2OPTCUDA* and *MLS2OPTCUDA* algorithms on the *P100*

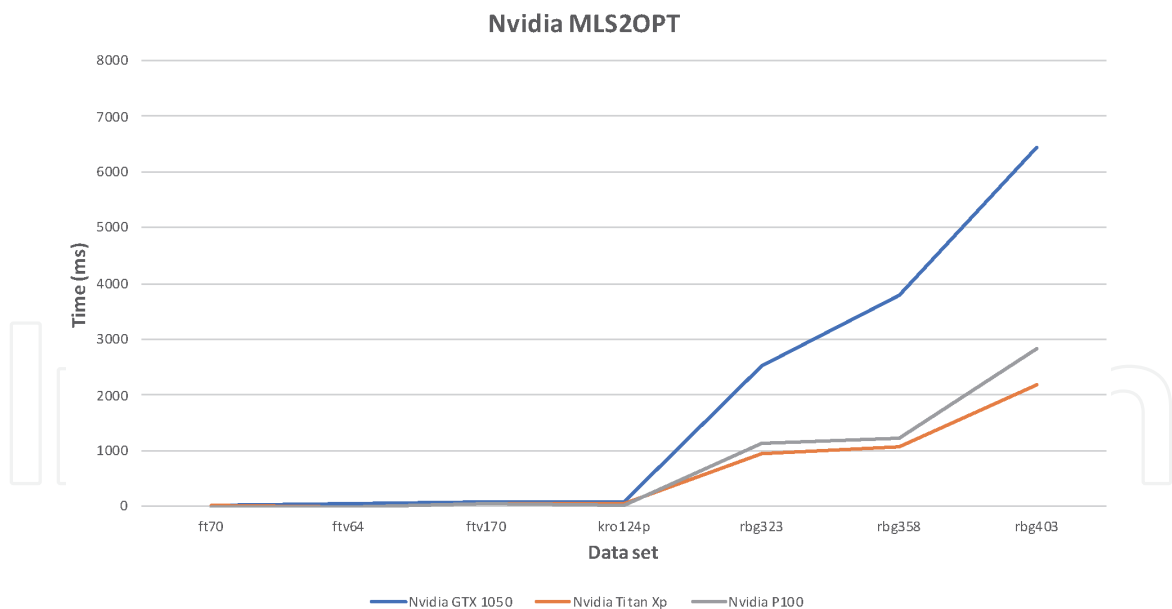


Figure 11.
Figure for the experiments of the three NVidia GPU's for the MLS2OPTCUDA algorithm.

Data	Optimal	LS2OPT	PRD (%)	MLS2OPT	PRD (%)
ftv70	38,673	43,163	−10.40	43,310	−10.71
ftv64	1839	2744	−32.98	2554	−28.00
ftv170	2755	4559	−39.57	4510	−38.91
kro124p	36,230	58,014	−37.55	55,011	−34.14
rbg323	1326	1681	−21.12	1535	−13.62
rbg358	1163	1625	−28.43	1459	−20.29
rbg403	2465	2710	−9.04	2598	−5.12
Average	12064.43	16356.57	−25.58	15853.86	−21.54

Table 9.
Comparison of 2OPT vs. optimal values.

GPU. The results are compared using the PRD Eq. (3). The GPU is replaced with the optimal value and the CPU is replaced by the obtained result.

The PRD values comparison shows that the LS2OPT is at most 40% away from the optimal value for *ftv170* instance and − 9% for the *rbg403* instance. For the MLS2OPT comparison, the PRD is −39% from the optimal value for *ftv170* instance and − 5% for the *rbg403* instance. On average, the MLS2OPT is a better performing algorithm with an average of 15853.86 against 16356.57 for the LS2OPT algorithm. A plot of the comparison values is given in **Figure 12**.

The second comparison is now done with four different evolutionary algorithms as given in **Table 10**. Theses are the Discrete Particle Swarm Optimization (DPSO) algorithm [37], Discrete Self-Organizing Algorithm (DSOMA) [38], Enhanced Differential Evolution (EDE) algorithm and the Chaos driven Enhanced Differential Evolution (*EDE_C*) algorithm [17]. The DPSO and DSOMA algorithms were revised for the TSP problem and the 2-OPT local search was removed from the algorithms to compare the results without any local search implemented. EDE and *EDE_C* are published algorithms however only three instances were published. Both these algorithms had the 2-OPT local search embedded in them.

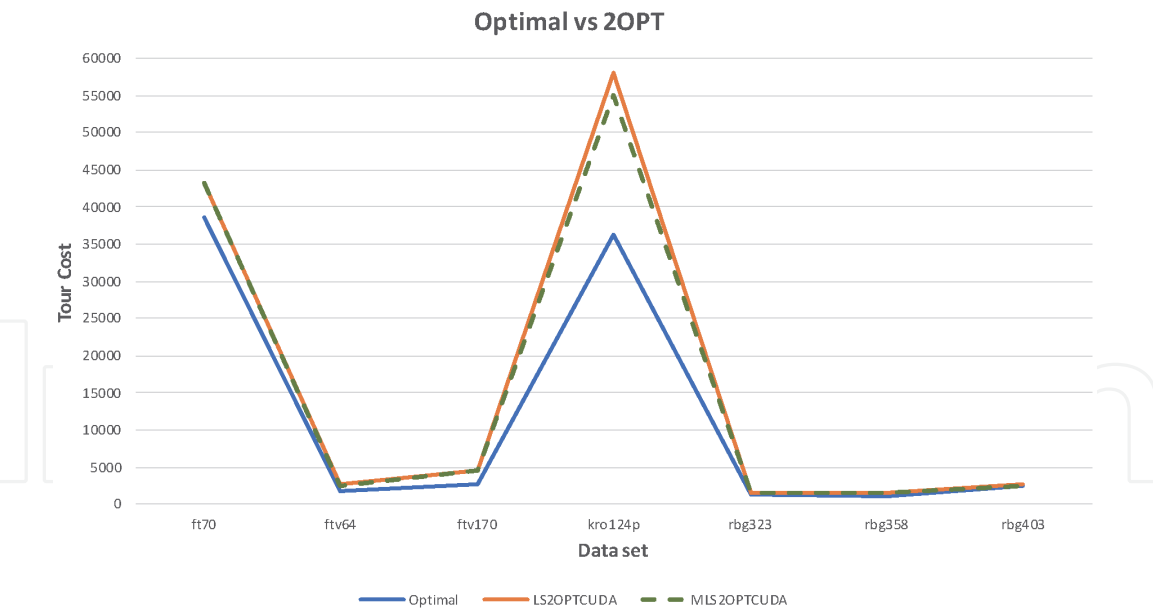


Figure 12. Figure for the comparison of 2-OPT against global optimal values [36].

Data	MLS2OPT	DPSO	DSOMA	EDE	EDE _C
ft70	43,310	54,444	51,325	40,285	39,841
ftv64	2554	4711	4423	—	—
ftv170	4510	19,102	9522	6902	4578
kro124p	55,011	113,153	75,373	41,180	39,574
rbg323	1535	4852	4523	—	—
rbg358	1459	5692	4874	—	—
rbg403	2598	6373	4427	—	—
Average	15853.86	29761.00	22066.71	—	—

Table 10. MLS2OPT vs. evolutionary algorithms.

From the results, it was obvious that evolutionary algorithms without local search heuristics are not as effective as the 2-opt local search heuristic or algorithms with both directed and local search combined. Therefore, it is important to combine these two algorithms as in [39]. As reported in [39] that the execution time of local search can be around 95–99% of the total run time of the algorithm, it is viable to accelerate the local search heuristics.

10. Conclusions

This chapter introduces a CUDA accelerated 2-opt algorithm for the TSP problem. As one of the most common and widely used approaches to solve the problem, the 2-opt approach can be considered as canonical in the field.

GPU programming, especially CUDA has gained significant traction for high performance computing. Readily available hardware has made programming a much easier and available task.

Two variants of the 2-opt algorithm have been coded in CUDA to show the acceleration of computational time. This has been tested against a sample of test

instances from literature. From the results obtained, it is clear that even for a relatively cheap GPU such as the GTX 1050 the performance improvement is significant, especially for larger sized problem instances. These were compared against industry leading CPU's such as Intel i7-X series and IBM Power 8.

One of the interesting aspects was that the Titan Xp performed better than the P100 for these instances. It is difficult to identify the reasons, as the same code was deployed on all machines, however the IBM and Intel architecture differences and different C/C++ compiler usage may have affected the performance. The physical configuration of the GPU's inside the hardware and its connection to the motherboard and memory bandwidth issues could also add to the time overhead. However, when analyzing the cost-performance of the GPU's then the \$1500 Titan Xp is a better GPU than the \$15,000 P100 in this case.

However, the clear distinction is that there is a significant improvement to be had when applying the CUDA version of the 2-opt algorithm. The next direction of this research is to combine it with powerful swarm meta-heuristics with a layered approach, and try and solve very large TSP instances.

Author details

Donald Davendra^{1*†}, Magdalena Metlicka^{2†} and Magdalena Bialic-Davendra^{1†}


1 Central Washington University, Ellensburg, USA

2 Honeywell Engineering Aerospace, Brno, Czech Republic

*Address all correspondence to: donald.davendra@cwu.edu

† These authors contributed equally.

IntechOpen

© 2020 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution License (<http://creativecommons.org/licenses/by/3.0>), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. 

References

- [1] Englert M, Roglin H, Vocking B. Worst case and probabilistic analysis of the 2-opt algorithm for the TSP. *Algorithmica*. 2014;**68**:190-264
- [2] Van Leeuwen J, Schoon A. Untangling a traveling salesman tour in the plane. In: *Proceedings of the 7th International Workshop on Graph-Theoretical Concepts in Computer Science*. The Netherlands: Rijksuniversiteit. Vakgroep Informatica; 1981. pp. 87-98
- [3] Johnson D, McGeoch L. The traveling salesman problem: A case study in local optimization. In: Aarts E, Lenstra J editors, *Local Search in Combinatorial Optimization*. Hoboken, NJ, USA: John Wiley and Sons; 1997
- [4] Farber R. *CUDA Application Design and Development*. Burlington, MA, USA: Morgan Kaufmann; 2012
- [5] Lawler EL, Lenstra JK, Kan AR, Shmoys DB. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Vol. 3. New York: Wiley; 1985
- [6] Davendra D, editor. *Traveling Salesman Problem, Theory and Applications*. Rijeka: IntechOpen; 2010
- [7] Laporte G. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operations Research*. 1992; **59**(2):231-247
- [8] Li H, Alidaee B. Tabu search for solving the black-and-white travelling salesman problem. *Journal of the Operational Research Society*. 2016; **67**(8):1061-1079
- [9] Kirkpatrick S, Gellat C, Vecchi M. Optimization by simulated annealing. *Science*. 1983;**220**(4598):671-680
- [10] Grefenstette J, Gopal R, Rosmaita B, Van Gucht D. Genetic algorithms for the traveling salesman problem. In: *Proceedings of the First International Conference on Genetic Algorithms and their Applications*. New Jersey: Lawrence Erlbaum; 1985. pp. 160-168
- [11] Oliver I, Smith D, Holland JR. Study of permutation crossover operators on the traveling salesman problem. In: *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms*; July 28–31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA, Hillsdale, NJ, L. Erlbaum Associates; 1987
- [12] Yu B, Yang Z-Z, Yao B. An improved ant colony optimization for vehicle routing problem. *European Journal of Operations Research*. 2009; **196**(1):171-176
- [13] Tang K, Li Z, Luo L, Liu B. Multi-strategy adaptive particle swarm optimization for numerical optimization. *Engineering Applications of Artificial Intelligence*. 2015;**37**:9-19
- [14] Yang X-S, Deb S. Cuckoo search via levy flights. In: *World Congress on Nature & Biologically Inspired Computing*, 2009. NaBIC 2009. NY, USA: IEEE Publications; 2009. pp. 210-214
- [15] Yang X-S. Firefly algorithms for multimodal optimization. In: *Stochastic Algorithms: Foundations and Applications*. Berlin, Heidelberg, Germany: Springer; 2009. pp. 169-178
- [16] Osabaa E, Sera D, Sadollah A, Miren Nekane Bilbaob J, Camachoe D. A discrete water cycle algorithm for solving the symmetric and asymmetric traveling salesman problem. *Applied Soft Computing*. 2018;**71**: 277-290

- [17] Davendra D, Zelinka I, Senkerik R, Bialic-Davendra M. Chaos driven evolutionary algorithm for the traveling salesman problem. In: Davendra D, editor. *Traveling salesman problem*. Rijeka: IntechOpen. DOI: 10.5772/13107
- [18] Li L, Cheng Y, Tan Y, Niu B. A discrete artificial bee colony algorithm for TSP problem. In: *Proceedings of the 7th International Conference on Intelligent Computing: Bio-Inspired Computing and Applications (ICIC'11)*. Berlin, Heidelberg: Springer-Verlag; 2011. pp. 566-573. DOI: 10.1007/978-3-642-24553-4_75
- [19] Davendra D, Zelinka I, Pluhacek M, Senkerik R. DSOMA—Discrete self-organising migrating algorithm. In: *Self-Organizing Migrating Algorithm: Methodology and Implementation*. Berlin, Heidelberg, Germany: Springer; 2016. pp. 51-63
- [20] Croes G. A method for solving traveling-salesman problems. *Operations Research*. 1958;**6**(6):791-812
- [21] Shen L. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*. 1965;**44**(10): 2245-2269
- [22] Savelsbergh M. An efficient implementation of local search algorithms for constrained routing problems. *European Journal of Operational Research*. 1990;**47**(1):75-85
- [23] Johnson D, McGeoch L. The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization*. 1997;**1**: 215-310
- [24] Gutin G, Punnen A. *The Traveling Salesman Problem and Its Variations*. Vol. 12. Berlin, Heidelberg, Germany: Springer; 2002
- [25] Kim B, Shim J, Zhang M. Comparison of tsp algorithms. In: *Project for Facilities Planning and Materials Handling*. 1998
- [26] Kronos Group. Available from: <https://www.khronos.org/> [Accessed: 24 May 2020]
- [27] Sanders J, Kandrot E. *CUDA by Example*. 1st Print Edition. Addison-Wesley; 2010
- [28] Kirk D, Wen-mei W. *Programming Massively Parallel Processors: A Hands-on Approach*. Newnes; 2012
- [29] NVIDIA: *Cuda C Programming Guide*. Santa Clara, CA, USA: NVIDIA Corporation; 2020
- [30] NVIDIA: *Kepler gk110*. Santa Clara, CA, USA: NVIDIA Corporation; 2012
- [31] Metlicka M. *Framework for scheduling problems [master thesis]*. Czech Republic: Technical University of Ostrava; 2015
- [32] Metlicka M, Davendra D, Hermann F, Meier M, Amann M. GPU accelerated NEH algorithm. In: *2014 IEEE Symposium on Computational Intelligence in Production and Logistics Systems (CIPLS)*, Orlando, FL; 2014. pp. 114-119. DOI: 10.1109/CIPLS.2014.7007169
- [33] NVIDIA. *Cuda C Best Practices Guide [Online]*. 2020
- [34] TSP Library ATSP Dataset. Available from: <http://elib.zib.de/pub/mp-testdata/tsp/tsplib/atsp/index.html> [Accessed: 02 January 2020]
- [35] CWU Turing Supercomputer. Available from: <http://www.cwu.edu/faculty/turing-cwu-supercomputer> [Accessed: 20 February 2020]
- [36] TSP Library ATSP Best Known Solutions. Available from: <http://elib.zib.de/pub/mp-testdata/tsp/tsplib/atsp-sol.html> [Accessed: 17 May 2020]

[37] Wang X, Tang L. A discrete particle swarm optimization algorithm with self-adaptive diversity control for the permutation flowshop problem with blocking. *Applied Soft Computing*. 2012;**12**:652-662

[38] Davendra D, Bialic-Davendra M. Discrete self organizing algorithm for pollution vehicle routing problem. In: *Proceedings of the genetic and evolutionary computation conference 2020 (GECCO 20 companion)*. New York, NY, USA: ACM; 2020. p. 8. DOI: 10.1145/3377929.3398076

[39] Merz P, Freisleben B. Genetic local search for the TSP: New results. In: *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97)*, Indianapolis, IN, USA; 1997. pp. 159-164. DOI: 10.1109/ICEC.1997.592288