Solving TSP on the GPU based on heuristic algorithms

Jóhann Utne, dvl176 - Henriette Naledi Winther Hansen, fvd409

Associate Professor: Cosmin Eugen Oancea

Department of Computer Science, June 10, 2023

Contents

1 Abstract

This project aims to efficiently solve the Travelling Salesman Problem (TSP) on the GPU. A short survey of three popular heuristic algorithms (2-Opt local search, Ant Colony Optimization, and Simulated annealing) is made. 2-Opt was chosen to be parallelized and implemented in CUDA. The 2 opt algorithm uses "climbers" that start with an initial solution, and improves it until a local optimum has been found. The GPU implementation exploits both degrees of parallelism by assigning one climber to each block and using threads to generate improved solutions. The code is publicly available at `https://github.com/henni3/BA-project`.
The solution is compared with another 2-Opt implementation on the GPU by O'Neil et al. [21] where one climber is assigned one thread, only using the outer parallelism. The project's solution was also evaluated on the accuracy, based on climbers and utilization of the hardware GPU measured against the big throughput.
Compared to the implementation by O'Neil et al., this project achieved up to 27 times speed up on a 100-city data set with 1000 climbers but falls off for greater numbers of climbers. The accuracy of the solutions was also within $< 5\%$ of the optimal solution for 1000 climbers, indicating that a greater number of climbers for some data sets might not be necessary.
Our implementation does not reach the big throughput of the GPU of 1555 GB/s for smaller data sets with < 100 cites, but for bigger data sets, the implementation exceeds the big throughput, since the majority of memory accesses happen to faster L2 cache. This shows that the implementation fully utilizes the GPU.

2 Introduction

The traveling salesman problem (TSP) is a well-known combinatorial problem, where a salesman must visit a set of n cities $\{c_1, ..., c_n\}$ [16], starting from one city, and then visiting each city one, returning to the starting city, creating a path. This path is known as a Hamiltonian path [33].
The goal of TSP is to create a Hamiltonian path, that minimizes the length of the path. To be able to solve this, it is required that each city pair has a distance, denoted $d(c_i, c_j)$ for the cities i and j. This project works with the symmetric TSP, meaning that

$$d(c_i, c_j) = d(c_j, c_i)$$

The problem can now be formalized, to finding a Hamilton path π, such that it minimizes the following equation

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) \ + d(c_{\pi(n)}, c_{\pi(1)})$$

where $c_{\pi(i)}$, refers to ith city on the path.
The reason the symmetric TSP is worked on is that it simplifies the problem instance and makes each easier to reason about, while still having many applications, such as X-ray crystallography.
Symmetric TSP is NP-hard, meaning no algorithm for finding optimal in polynomial time has been found. One way to find the optimal solutions is to find every possible path, and then take the path with minimum length. This would require $n!$ permutations and is infeasible to compute for any meaningful problem sizes.
Therefore meta-heuristic algorithms are popular to solve TSP [25]. These algorithms usually find

approximations of the optimal solution or local optimums of the solutions. Some popular ones are the local search approach, such as the 2-Opt algorithm [6], the evolutionary approach such as Ant Colony Optimization (ACO) [24], and others inspired by real-life processes, like Simulated Annealing(SA) [30].
These algorithms start with some initial solutions and work towards optimizing them. The 2 opt algorithm uses an iterative hill climber (IHC) approach, where it iteratively finds a change, by swapping two edges, that improves the solution until no further improvements can be found. At this point, a local optimum has been found. Random restarts are then used to circumvent being stuck in a local optimum
Evolutionary algorithms like ACO use a population-based approach instead. ACO is inspired by how ants find a good path to food, by leaving pheromones. The shortest path gets more pheromones, which creates a positive feedback loop. ACO starts with an initial population of ants, which finds a path, where each edge is traveled according to some probability, based on the amount of pheromones and the length of cost of the edge. When each ant has found a path, the pheromones are updated based on the length of the paths found, and new ants repeat the process.
SA uses an annealing process, where a temperature parameter controls how likely the algorithm is to explore changes that decrease the quality of the solution. This makes it possible to get out of local optimums, to potentially find the global optimum.

For these algorithms, computing and evaluating changes requires many computations, and can get very computationally expensive. Most of the computations can be done independently, however, e.g. a restart in the 2-opt algorithm, does not depend on any other restarts. Furthermore, the interchange evaluations are also independent. For ACO, each ant can create a path independently, for each iteration. This indicates that a lot of the work is parallel.
For this reason, this project will solve TSP on GPU, since the GPU is ideal for problems with a lot of parallelisms, and low communication. This is due to its thousands of cores, compared to the low counts of the CPU.
In this project, the 2-Opt algorithm has been implemented on the GPU, due to its parallel structure. The implementation was firs written in the parallel language Futhark, but because of limitations, was then written in the language CUDA. A distance matrix that fits in L2 memory is created, and each climber is mapped to one block. In the block, each thread evaluates 2-opt changes.
The implementation is compared to another 2-Opt on GPU version by O'Neil et al, where each thread handles one climber, and the distance matrix fits in shared memory. [21].
From this, the following is found

- For a smaller number of climbers ($\leq$ 10.000), our version achieves up 27 times speedup
- A small number of climbers is needed to find a solution with $< 5\%$ of the optimal cost for most data sets
- Putting the distance matrix in shared memory, only achieves 1.11 times speed up compared to L2 memory, but limits the problem size to 100 cities

Since our implementation is memory bound, it was also measured how well it utilized the hardware GPU, compared to the big throughput, measured in GB/s. It was found that for smaller data sets $< 100 cities$, the GPU is not fully utilized, however for bigger data sets > 100, the throughput exceeds the big throughput. This happens since the majority of data accesses happen to distance matrix that fits in L2 memory, which is faster than global memory.

The project is structured the following way.

- section 3 surveys and explains the heuristic algorithms, some GPU implementations of different algorithms, and why the 2-Opt algorithm was chosen.
- section 4 is a user guide for the code base and repository
- section 5 describes how the sequential implementation of 2-Opt was parallelized and implemented in Futhark
- section 6 explains how the algorithm was implemented in CUDA
- section 7 evaluates the sequential versions of the heuristic algorithms, evaluates our implementation, and compares it to the implementation by O'Neil et al. [21]

3 Related works

This section will go over three popular approximation algorithms for solving TSP. The algorithms that will be described and analyzed are the 2-Opt local search algorithm, Ant Colony Optimization, and simulated annealing.
Then there is a discussion of why the 2-Opt-algorithm was chosen to implement parallelly on the GPU.
Finally a description of other GPU implementations of heuristic algorithms is made.

3.1 2-Opt local search algorithm

The 2 opt local search algorithm is also known as a local search strategy, that implements the iterative hill climbing strategy (IHC)

IHC is a local search strategy that takes an initial solution to a problem and then does incremental change to try and improve the solution. This is like starting at the base of a hill, and then gradually climbing to the top, hence the name. This only guarantees a locally optimal solution, without finding a global solution. To get a better chance of finding the optimal solution, random restarts are made.

The 2-Opt-move algorithm is one of the simplest optimization methods for TSP proposed by Croes [6]. The 2 opt algorithm starts by creating an initial tour as a solution. It then finds some interchange of 2 edges of the tour such that the cost of the tour decreases. The interchange is evaluated by interchanging the edges $e(i, i+1)$ $e(j, j+1)$such that the new edges are $e(i, j)$, $(i+1, j+1)$. All edges between i and j, are then reversed, to accommodate this change. E.g the path $[1, 2, 3, 4, 5, 6, 7, 1]$ when $i = 3$ and $j = 6$ would become $[1, 2, 6, 5, 4, 3, 7, 1]$ This interchange is shown in the figures 1 and 2.
If this swap results in a better tour, the incremental change is kept and the solution is updated. Otherwise, the change is discarded and a new interchange will be found.

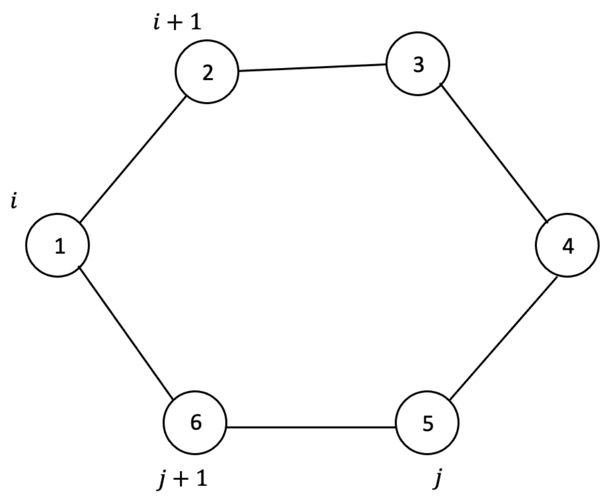


Figure 1: Before swapping

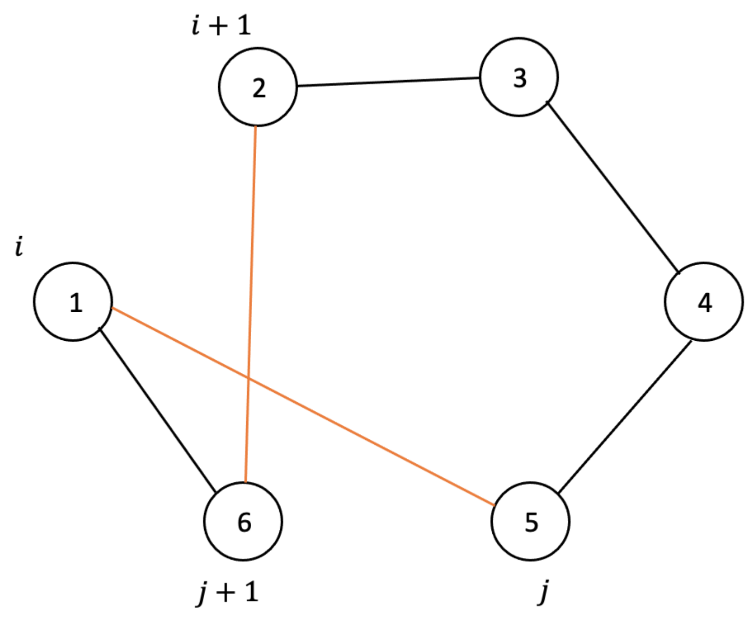


Figure 2: After swapping

To find this interchange, all interchanges are evaluated in a double for-loop. Then only the best interchange is applied to the solution before a new one is found.
To not be stuck in a local optimum a strategy of random restarts is used. Here multiple random

initial solutions are created, such that each finds a local optimum. By having enough random initial tours, it is probable that one of these local optima is the real optimal solution or a local optimum close to the global.

3.1.1 Algorithm

Since this project implements the 2-Opt local search algorithm, A detailed pseudo-code has been provided.

```
//tour, is the array that holds the cities in a random order
//(starting and ending in the same city)
//distM, is the distance matrix between the cities
findCheapestCost(distM, cities, climbers){
    int [] bestTour = initializeTour();
    int bestCost = infinite
    for (int c = 0; c < climbers; c++){
        int[] tour = createTour(cities);
        int minChange = -1;
        while(minChange < 0){
            minChange = 0;
            // 2 opt move
            for(int i = 0; i < (cities - 2); i++) {
                for(int j = i+2; j < cities; j++){
                    change = distM[tour[i]][tour[j]] +
                    distM[tour[i+1]][tour[j+1]] -
                    (distM[tour[i]][tour[i+1]] +
                    distM[tour[j]][tour[j+1]]);
                    if(change < minChange){
                        minChange = change;
                        min_i = i;
                        min_j = j;
                    }
                }
            }
            // swap
            if(minChange < 0){
                min_i = min_i + 1;
                while(min_i < min_j){
                    to = tour[min_j];
                    tour[min_j] = tour[min_i];
                    tour[min_i] = to;
                    min_i ++;
                    min_j --;
                }
            }
        }
        int cost = 0;
        for(k = 0; k < cities; k++){
            cost += distM[tour[k]][tour[k+1]];
        }
        if(cost < bestCost){
            bestTour = tour;
            bestCost = cost;
        }
    }
    return (bestCost, bestTour);
}
```

The algorithm evaluates one climber at a time. When one climber has found a solution providing a local optimum, it compares with the previous best-known solution. If the newfound solution was better, the solution and lowest cost are updated. The solutions are evaluated on the length of their tour. The main work of the algorithm is the double for loop which has a run time of $O(cities^2)$. this is then repeated for m iterations, where m is a number of improvements that need to be made before a local optimum is found. Lastly, all this work is done for each worker giving a run time of $O(m \cdot climbers \cdot cities^2)$.
This is quite the workload, however, a lot of it can be parallelized and computed by the GPU, this will be discussed further in the project.

3.2 Ant Colony Optimization

3.2.1 General description

Anthill Colony Optimization (ACO) is a popular approximation algorithm used to solve TSP. The algorithm is inspired by ants in the real world, where ants create optimal paths from their colony to a food source. This happens because ants leave a trail of pheromones when traversing a path. Therefore when an ant with a shorter route returns from the food, the pheromones on the path are reinforced faster than the other paths. This then gets more ants to follow the short path, reinforcing it for future ants. The pheromones evaporate over time, so only the more traversed path over time is taken by the ants. [9]

This method can be translated to TSP with all possible paths as the search space for the ants. Each ant starts at a city and then creates a Hamilton cycle, by going from city i to city j, according to a transition formula, until a path is created. When tours consisting of all cities have been found by the ants, the pheromones are on the edges are updated. The update is based on the total length of a tour, where a shorter tour will result in a stronger pheromone update.
The probability that an ant transitions from one city to another will be dependent on the pheromone on the edge and heuristic information. The heuristic information for the TSP problem will be inversely proportional to the cost, such that edges with lower costs will give better probabilities. This makes the ants more likely to traverse edges which results in a lower cost.
There is no specific rule for when the algorithm should converge and multiple strategies can be implemented. One simple and popular choice is to run a specific number of iterations and then evaluate what the best path is [24].

3.2.2 The transition probability formula

A general strategy for computing the probability of an ant choosing an edge is [29]

$$\frac{|\tau_{i,j}^{\alpha}|\eta_{i,j}^{\beta}|}{\sum_{z\in A_i}\tau_{i,z}^{\alpha}\eta_{i,z}^{\beta}}$$

Where

- $\tau_{i,j}$ is the pheromones on edge e(i,j)
- $\eta_{i,j}$ is the heuristic information on edge e(i,j)
- α is a factor determining the strength of the pheromone
- β is a factor determining the strength of the heuristic information
- A_i is the set of all nodes that can be reached from edge i and is not visited by the ant yet

3.2.3 Pheromone update formula

There exist different strategies to update the pheromones. Two popular strategies are :

- The Min-Max Ant system, where only the ant that found the best path updates the pheromones. The pheromones on a path can also not decrease below a τ_{min} or above a τ_{max}.
- The Ant System rank where all pheromones are evaporated by a factor with variable ρ and then every ant leaves a quantity of pheromone Q divided by the length of the tour the ant found $\frac{Q}{Lentgh_{ant}}$

The Ant system Rank version is the version that will be used in this project. The pheromones update for edge (i,j) can be written as the formula

$$\tau_{i,j}^{'} = \rho \cdot \tau_{i,j} + \Delta\tau_{i,j}$$

Where $\Delta\tau_{i,j}$ is the total amount of pheromones laid by all ants that traversed this edge and can be expressed as

$$\Delta\tau_{i,j} = \sum_{a=1}^{Maxants} \Delta\tau_{i,j}^{a}$$

and

$$\Delta\tau_{i,j}^{a} = \begin{cases} \frac{Q}{Length_a} & \text{if ant k travels on edge (i,j)} \\ 0 & \text{otherwise} \end{cases}$$

Where

- Q: is the chosen amount of pheromones laid by an ant
- ρ is the evaporation factor
- Maxants is the number of ants traversing paths

- $\Delta\tau^{a}_{i,j}$ is the pheromone change on edge (i,j) by ant a
- $\Delta\tau_{i,j}$ is the total change of pheromones by all ants

the parameters Q, α, β, ρ, and the number of ants are chosen before running and should be optimized for the specific problem.

3.2.4 Parameters in ACO

The parameters have a big influence on the performance of the solution. If the pheromone factor *alpha* is too big, the solution might converge too quickly. This leads to not enough solutions being searched, making it unlikely to find an optimal solution.
The same argument can be made for the evaporation factor ρ, since if the pheromones on the edges evaporate too quickly, the ants quickly disregard them. The amount of pheromones Q and the heuristic factor β also influence the quality of the algorithm, since the β parameter control the scope of the search space, by the heuristic information. An article from Bullnheimer et.al [4] suggests that α should be lower than β because the heuristic information should have a greater influence. α should not be too low, because otherwise, the feedback loop from the pheromones is not exploited. It is found that the Q factor is not too important as long as it is not too low.
Studies suggest that the number of ants should be close to the number of cities and start in random different positions [24].

These parameters are very important for the quality of the solution and should be optimized for each problem they are used for. Parameter tuning is a big part of the ACO algorithm to find good solutions and should be done for the specific problem.

3.2.5 Algorithm

To compare the ACO algorithm with 2-Opt local search, a simple solution based on the work of [20] was created in c. The algorithm will only be described in pseudocode, as it is not studied much further in this project.

```
ACO(int ants, string dataSet, int iterations) {

    int** DistMatrix = createDistMatrix(dataSet);
    Q = 100.0;
    alpha = 1.0;
    beta = 5.0;
    rho = 0.5;
    int** startPaths = intializeAnts(ants, DistMatrix, lengthOfTour);
    double** Pheromones = initphero(lengthofTour)
    int BestCost = findBestPath(ants, distMatrix, lengthOfTour)
    for (int i = 0; i < iterations; i++
        UpdateAnts(startPaths, lengthOfTour, Pheromones, distMatrix)
        UpdatePheromones(Pheromones, startPaths, DistMatrix, lengthOfTour)
        int newCost = findBestPath(startPaths, DistMatrix, lengthOfTour);:
        if (newCost < BestCost) {
            BestCost = NewCost
        }
    return BestCost0
}
```

The parameters are chosen in accordance with the findings of Wei in the article "Parameters Analysis for Basic Ant Colony Optimization Algorithm in TSP" [32]

3.3 Simulated Annealing

3.3.1 General description

Simulated annealing is another popular optimization heuristic method to solve optimization problems like TSP. Simulated annealing is stochastic, and uses a temperature factor, to generate random moves, to avoid getting stuck in a local optima. The method was introduced by Kirkpatrick et al [19], who was inspired by the annealing process of metallurgy. Annealing consists of heating material to high temperatures to change the properties of the metal.

3.3.2 Algorithm description

Simulated annealing builds on the annealing idea by having a temperature parameter (T), that starts high and gradually cools over time. At higher temperatures, there is a bigger probability to accept changes, which worsens the quality of the solution. At lower temperatures, only changes that improve the quality of the solution are likely to be made. The acceptance of changes that leads to a deterioration of quality, is done, to avoid being stuck in a local optima. Each change is called a transition to a neighboring solution. The neighborhood solution can be obtained with multiple strategies, a simple one for TSP problems is doing a 2-Opt swap, as described in 3.1. If the change results in a better solution, it is always accepted. If the change worsens the quality of the solution, it is accepted with a probability, that depends on the temperature and the cost of the change. The probability is based on the Metropolis criterion [7] The state transition probability can be expressed by the formula [9]

$$P(s') = \begin{cases} 1 & \text{if } f(s') < f(s) \\ e^{(-\frac{(f(s')-f(s))}{T}} & \end{cases}$$

Where

- s' is the new solution obtained by generating the neighboring solution
- s is the original solution, from which the neighboring solution was made
- f(s) is the objective function, that evaluates the quality of a solution
- T is the value of the temperature, at the given transition.

To evaluate this probability, usually, a value p is assigned to the transition probability $p = P(s')$, and then another value c is randomly generated such that $0 < c < 1$. The solution s' is accepted if

$$c < p$$

After each transition, the temperature is cooled, determined by a cooling schedule. Many different cooling schedules exist such as logarithmic, linear, exponential, and more [23]. The exponential schedule is widely used and decreases the temperature T with a factor α $0 < \alpha < 1$ expressed as

$$T(t) = T_0\alpha^t$$

where

- T(t) is Temperature at time or iteration t
- T_0 is the starting temperature
- α^t is a cooling factor to the power of the iteration

Since every other parameter but the temperature is predetermined, and only temperature changes, the choice of cooling schedule is critical.[23].

The algorithm terminates when a condition is met. The condition is usually a low enough temperature is reached, or a desired number of iterations is done.

3.3.3 Simulated Annealing and TSP

To relate this to the TSP problem, the simulated annealing method starts with an initial solution. The initial solution will be a random permutation of cities of a data set, representing a path. The neighboring solutions will be generated, by making 2-Opt moves. The path will be evaluated with a fitness function $f(path)$ which is the cost of the path. After a neighboring solution has been generated it is accepted according to the metropolis criterion. Then the temperature is decreased according to a cooling schedule.
A pseudocode for simulated annealing in tsp is shown below. T

```
Simulated annealing(double T_0, double alpha, char* data_set)
    int[] path = createRandomPath(data_set)
    double T = T_0
    while (T < stop_cond) {
        int City_i = rand(Cities)
        int city_j = rand(Cities)
        if( i != j) {
            new_path = 2OptSwap(path, i , j)
        }
        if(f(path) > f(new_path) {
            path = new_path
        }
        else {
            double p = exp(-(f(new_path) - f(path))/T)
            double c = rand(0,1)
            if (c < p) {
                path = new_path
            }
        }
    T = T * pow(alpha, t)
    }
```

3.3.4 Discussion of simulated annealing

Simulated annealing is very reliant on good choices for starting temperature, and cooling schedule. If the problem has many local optima, a high temperature, and a slower cooling schedule will be needed, to ensure a global optimum is found. This however affects running time and will converge very slowly. However, if the starting temperature is too low or the cooling schedule cools too much, the algorithm can quickly be stuck in a local optimum. The temperature and cooling schedule should be explored and chosen for the specific problem being evaluated.

3.4 Choice of a heuristic algorithm for the GPU

For our project, we have chosen to parallelize the 2-Opt local search, with random restarts for the GPU. The parallelization of the sequential code will be explained in section 5. This choice will be explained in the following subsections.

3.4.1 Fitness for GPU

The 2-Opt local search algorithm with random restarts is a good candidate for the GPU since each climber with a random initial solution can be computed independently of other climbers, and therefore also parallelly. Furthermore, each swap can also be computed independently, before a swap is made. The selection of the best swap cannot be done independently but can be parallelized differently. This is discussed more in detail in section 5 5.

For the Ant Colony Optimization (ACO) each ant is able to compute a path independently since it only depends on the pheromone table of the previous iteration. After each iteration, however, the pheromone table needs to be updated, which requires communication. ACO also requires much more space as a pheromone matrix is required in addition to a distance matrix and tour list for each ant. The pheromone matrix has the same space requirement as the distance matrix for TSP problems and scales quadratically with problem sizes.
SA however does not show good promise for parallelization, as only one neighbouring solution is generated and evaluated at a time. The while loop is dependent on the previous iteration and each transition needs to be taken with some probability dependent on the current temperature and fitness of the change.

3.4.2 Parameters

Since both ACO and 2-Opt Local search, shows promise for parallelization, the biggest factor for choosing the 2-Opt local search algorithm, with random restarts, is that it does not depend on parameters that need to be optimized for specific problems. Both the Ant Colony Optimization (ACO) and simulated annealing algorithm relies heavily on good choices for the parameters to give good results. 2-Opt local search is, therefore, better suited for general problems, since no parameters needed to be determined. The 2-Opt local search also always finds a local optimum, while ACO or SA might terminate before finding one, depending on the stopping criteria. Since both ACO and SA can be stopped with a number of iterations, they might stop, before finding any optimum, resulting in a bad-quality solution.

3.4.3 Results of testing sequential versions

Sequential versions of the algorithms were made, to compare how quickly they converge to a solution, and the quality of the solution. The SA and ACO parameters were chosen based on findings from [24] [32] but were not tested further. The tests can be seen and are described in 7.3
The tests showed that 2-Opt quickly found a local optimum, and was the only one to find the global optimum. Neither SA nor ACO found the global optimum, and the solutions computed by them were worse than the first local optimum found by the 2-Opt.

For these reasons, the 2-Opt local search algorithm has been chosen to be implemented on the GPU.

3.5 Other GPU implemenations

A rich body of work is aimed to solve TSP using approximation heuristics on the GPU. Molly A. O'Neil, Dan Tamir and Martin Buscher [21] created a CUDA implementation, where they use 2-Opt iterative hill climbing to solve traveling salesman problems with < 100 cities. They assign one climber to one thread to get a high degree of parallelism and minimize the need for synchronization. To handle load imbalance a worklist is used and threads terminate when the worklist is empty. They report good accuracy, where for four out of five data sets with 100 cities, the optimal solution was found. They report a speed-up of up to 62 times compared to a multi-threaded CPU solution.

Bai et. al. [3] proposes an Ant Hill colony implementation, of the Min-Max variation, for the GPU with multiple colonies. Each colony is mapped to a block, with its own pheromone matrix. All the colonies share a distance matrix. Each ant is mapped to one thread and finds a tour. A reduction is made to find the best tour, and the evaporation of pheromones in the pheromone matrix is done in parallel. Then, pheromones are deposited, where each thread deposits pheromones on the edges. The solution reports up to 32x speedup compared to a CPU solution of ACO, and the solutions found are close to the optimal solutions.

Wei et.al [31] proposes a Simulated Annealing implemented on the GPU. Here each block gets an initial solution and starts the annealing process. The parameters are user-defined and are the same as the one that was found to work for the sequential version analyzed. Each thread then finds a neighboring solution and evaluates whether or not they are feasible, according to the metropolis criterion. The GPU solution reported up to 14.8 times bigger speed compared to the sequential version and could find up 25% better solutions

These papers do not compare their GPU version directly to another GPU version, only to the CPU version of the same algorithm. In this project, we will compare our solution, with the solution made by O'Niel et. al, since both projects use the same approximation algorithm.

The projects defers in how the algorithms are implemented on the GPU. O'Neil et.al maps each climber to one thread and needs to have many climbers to fully satisfy the hardware GPU. The distance matrix is put in shared memory, giving fast access, but limiting the problem sizes.
Our solution maps each climber to a block and uses the threads inside each block to parallelize the local search of a climber, resulting in fewer climbers needed to satisfy the hardware GPU.
Our findings show that for most problems a small number of climbers is needed to find a good solution. It is also found that putting the distance matrix in shared memory results in an 11% speed increase, which indicates that it might not be advisable since it limits the problems that can be computed. Because most problem instances distance matrix fits in the L2 cache, which is faster to access than global memory, it results in fast enough execution.
The other GPU implementations were not compared, since the codebases were not publicly available, making it infeasible to compare meaningfully in this project.

4 User guide

This section consists of the following

- section 4.1 introduces the structure of the Github repository
- section 4.2 gives an overview of the project files and gives a short explanation of each file.
- section 4.3 describes how to compile and run the programs.

4.1 Structure of Github repository

The GitHub is publicly available `https://github.com/henni3/BA-project`, and the repository has the following directory tree

```
BA-project
├── CUDA
│   ├── TexasUni
│   └── CUDA_tests
├── Data
├── Futhark
└── Seqventielt
    └── Seq_test
```

- The CUDA directory contains the CUDA implementation of the 2-Opt algorithm, together with publicly available code from O'Neil et.al [21] [21]. It also contains the performance data from the performance tests on our version
- The Data directory, contains all the data sets acquired from the TSBLIB95 library [15]
- The Futhark directory contains the Futhark implementation of the 2-Opt algorithm
- The Seqventielt directory, contains the sequential implementations of the 2-Opt, ACO, and SA algorithms

4.2 File descriptions

In the following lists, there will be an overview and a short explanation of the project files.

4.2.1 Sequential project files

- `makefile` - compiles the sequential code implementations of TSP-ACO.c, TSP-SA.c, and TSP-seqRand.c
- `tsp.data.h` - provides functionality for reading TSP data sets
- `TSP-ACO.c` - contains the sequential implementation of Ant Colony Optimization
- `TSP-SA.c`- contains the sequential implementation of Simulated Annealing
- `TSP-seq.c` - contains the sequential implementation of 2-Opt local search without restarts
- `TSP-seqRand.c` contains the sequential implementation of 2-Opt local search with random restarts

4.2.2 Futhark project file

- `TSP.fut` - main program

4.2.3 CUDA project files

- `makefile` - compiles all the files
- `tsp-main.cu` - main program
- `tsp-run-program.cu` - sample program
- `tsp-testing-main.cu` - testing program (on different climbers)
- `tsp-gb-test.cu` - testing program (on GB/s)
- `tsp-main-helper.cu.h` - helper functions to the main programs
- `tsp-kernels.cu.h` - kernel functions from where version 1, 2, 3 and 4 are executed
- `tsp-ker-helper.cu.h` - kernel helper functions
- `dataCollector.cu.h` - collects and transforms the data input
- `constants.cu.h` - contributes with useful constants and functions
- `hostSkel.cu.h` - contributes with functions such as scan, segmented scan, and transpose
- `pbbKernels.cu.h` - contributes with kernel functions

4.3 Compile and run the programs

4.3.1 Sequential programs

To compile the code run

```
make all                // compiles all the versions
make tsp_ACO            // compiles the ACO implementation
make sim_ann            // compiles the SA implmenetation
make tsp_opt            // compiles the 2-Opt implementation
make clean              // removes the executables
```

To run the codes, different arguments need to be given. To run the ACO code

```
./aco <../Data/dataset.tsp> <number of ants> < number of iterations>
```

to run the SA code

```
./sa <../Data/dataset.tsp>
```

and to run the 2-Opt code

```
./2opt <../Data/dataset.tsp> <number of climbers>
```

4.3.2 Futhark program

To run the Futhark program, insert

```
futhark cuda TSP.fut
```

into the terminal to compile the program. Then insert

```
./TSP -t /dev/stderr -r 10 < ../Data/berlin52.txt
```

this will print the time 10 times and the final number (ending with i32) will be the best cost of the tour computed. When loading the dataset berlin52.txt it will automatically run with 100 climbers.

4.3.3 CUDA programs

There are four different main programs that have four different purposes.
To run a sample program insert

```
make pre_pro && ./pre_pro
```

to terminal. This program will run on the dataset `KroA100.tsp` with 75000 climbers.

To run the program on different datasets, different numbers of climbers, and on different versions (1: the Original Version, 2: 100 Cities Version, 3: Calculated I and J Version, see more in section 6.3), insert

```
make tsp && ./tsp ../Data/swiss42.tsp 1000 1
```

to terminal. `swiss42.tsp` can be exchanged to another dataset. 1000 can be modified to another number of climbers and 1 can be either version 1, 2 or 3.

To test the program on a dataset with different numbers of climbers insert

```
make test && ./test ../Data/KroA100.tsp 10 100000
```

to the terminal. This will run on the dataset `kroA100.tsp` with 10 climbers, then 100 climbers, then 1000 climbers, then 10000 climbers, and lastly 100000 climbers. So the minimum number of climbers (the first number) will run and then be multiplied by 10 and run, up to the maximum number of climbers (the second number). The dataset can be changed and the minimum and maximum number of climbers can be modified as long as it is a positive number and the minimum is smaller than the maximum.

To test the program on GB/s insert

```
make test_gb && ./test_gb 10000 ../Data/KroA100.tsp ../Data/swiss42.tsp
```

in terminal. 10000 is the number of maximum climbers, where the program will on $\frac{maximum\ climbers}{10}$, $\frac{maximum\ climbers}{9}$..$\frac{maximum\ climbers}{1}$ climbers for each of the datasets that are listed.

4.3.4 O'Niel et. al solution: TSP_GPU11

To run the O'Niel et. al solution [5] insert

```
make test-tsp11 && ./test-tsp11 ../../Data/kroA100.tsp 10000
```

in to the terminal. This will compile and run the program on the dataset `kroA100.tsp` with 10000 climbers. Both the dataset and the number of climbers can be modified.

5 Parallel program and Futhark Implementation

This section introduces

- section 5.1 gives a short overview of Futhark and important constructs.
- section 5.2 provides an analysis of the sequential code of the 2-Opt algorithm and explains how to transform the program into a parallel program. The parallel program is implemented in the Futhark language.
- section 5.3 introduces the limitations of writing the parallel program in Futhark.

5.1 Futhark and important methods

Futhark is a programming language made to write code that compiles efficient parallel code. Futhark is a purely functional language, meaning all parallelism is made by constructs such as maps, reduces, and scans. There are multiple papers that use Futhark to make solutions for the GPU[22][27][14]

5.1.1 Description of map

One of the most important, and most basic constructs is the map. The map is a second-order operator which takes the arguments: a unary function F and an array of elements $[x_1, ..., x_n]$. It produces an array of the same length as the input array, by applying the function on each array. map can be written with type and semantics:

$$map : (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$$

$$mapF[x_1, ..., x_n] = [fx_1, ..., Fx_n] \tag{1}$$

This is parallel as each element is computed independently of the other elements.

5.1.2 Description of reduce

The second important construct is the reduce second-order operator. It takes as an argument a binary and associative operator $\odot$, the neutral element of $\odot$, e, and an array of elements. Reduce applies the operator on all the elements successively, resulting in a single element
reduce can be written with semantics and type:

$$reduce : (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow \alpha$$
$$reduce : \odot e[x_1, ..., x_n] = e \odot x_1 \odot, ... \odot x_n$$

Reduce is parallel since it applies the reduce as a reduction tree, where each thread can reduce 2 elements each, and store the result. The new array is then reduced in the same manner, resulting in $log(n)$ parallel operations. This is also the reason why $\odot$ must be associative, otherwise, the parallel execution could give a different result than the sequential execution.

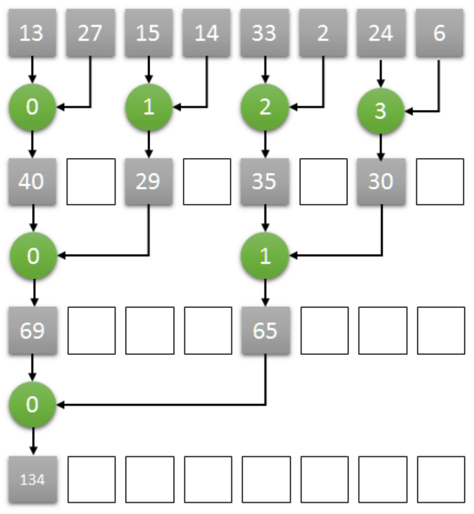


Figure 3: Parallel reduction tree, figure from Elemar Júnior [17]

5.2 From sequential to parallel

In the pseudo-code for the sequential implementation of the 2-Opt algorithm, see section 3.1.1, there are six loops in the program where all of the work is done. It is desirable to flatten as many loops as possible and in the following sections, each loop will be discussed. All the pseudo-code, in this section, for the parallel implementation is based on the Futhark language.

5.2.1 Outer most `for` loop in parallel

To begin determining which parts of the sequential pseudo code in section 3.1.1 can run in parallel the outermost `for` loop is analyzed first. This `for` loop has the following sequential pseudo code:
Sequential pseudo code of the outermost `for` loop

```
1  bestTour = initializeTour; bestCost = infinite;
2  ..  for (int c = 0; c < climbers; c++){
3          tour = createTour;
4          ... //compute local best tour
5          //compute cost
6          cost = 0;
7          for(k = 0; k < cities; k++){
8              cost = cost + distM[tour[k]][tour[k+1]];
9          }
10         //store the best cost and tour
11         if(cost < bestCost){
12             bestTour = tour; bestCost = cost;
13     }} return(bestCost, bestTour);
```

In the pseudo-code, there are cross-iteration dependencies. The dependency is for the variable `cost` in the inner `for` loop, line 7-9. `cost` is written in iteration i-1 and read in iteration i. This is a read-after-write (RAW) hazard which is equivalent to a true dependency. Since the inner loop that computes `cost` is basically a sum function it is possible to eliminate the dependency. Now that there are no cross-iteration dependencies the outermost `for` loop can safely execute in parallel. The pseudo-code for the parallel implementation is:
Parallel pseudo code of the outermost `for` loop

```
..  let allLocalBestCost = map(\climber ->
                                 let localTour = mkRandomTour
                                 ... --compute local best tour
                                 in cost localBestTour distM
                                )(iota numClimbers)
    in reduce costComparator maxInt allLocalBestCost
```

The outer `for` loop is transformed to run in parallel with a `map` and `reduce` function. The `map` function will compute all the local best cost of each climber in parallel and the `reduce` function will find the best cost of all the climbers and return this. Note that in the Futhark version, it is only the best cost that is returned.
For each climber, a random tour is computed and the local best tour is found. Then the cost of the local best tour is computed and this is done by transforming the `for` loop in line 7 of the sequential code above to a `map` and `reduce` function to avoid data dependencies. The `cost` function is:

```
let cost [n] [m] (tour : [n]i32) (distM : [m]i32) : i32 =
        map (\i -> distM[tour[i] * (n-1) + tour[i+1]]
            ) (iota (n-1)) |> reduce (+) 0
```

Note here that the addition (+) operator is both associative and commutative so it works well for the `reduce` function provided by Futhark.

To compute the best cost of all the climbers an `if`-statement is used in the sequential implementation, see line 11 in the sequential pseudo code above. Instead of the `if`-statement inside the `for` loop all the local best cost from each climber is stored into an array `allLocalBestCost` and a `reduce` function is used to find the best cost. The `reduce` function has the comparative operator:

```
let costComparator (cost1: i32) (cost2 : i32) : i32 =
    if cost1 < cost2 then cost1 else cost2
```

This compare operator, `costComparator`, is associative

```
changeComparator cost1 (changeComparator cost2 cost3) =
changeComparator (changeComparator cost1 cost2) cost3
```

and commutative

```
changeComparator cost1 cost2 = changeComparator cost2 cost1
```

It can therefore be used by the Futhark `reduce` function [8].

5.2.2 Outer `while` loop and `swap`

Inside the outermost `for` loop, discussed in section 5.2.1, there is a `while` loop that computes the local best tour and it has the following sequential pseudo code:
Sequential pseudo code of the `while` loop

```
1   ..  tour = createTour(cities);
2       while(minChange < 0){
3           minChange = 0;
4           for(..){for(..){change=distM[tour[i]][tour[j]]+...  //2 opt move
5               if(change < minChange){minChange = change;...}}}
6           if(minChange < 0){                                   // swap
7               min_i = min_i + 1;
8               while(min_i < min_j){
9                   to = tour[min_j]; tour[min_j] = tour[min_i];
10                  tour[min_i] = to; min_i ++; min_j --; }}}  ...
```

The `while` loop starting from line 2 can not be executed in parallel for two reasons. The first reason is that it is a convergence loop, that checks the previous iteration to see if a local optimum is found and it is therefore not possible to determine how many iterations the loop will execute. The second reason is that there are two cross-iteration dependencies in the `while` loop. The two variables that can cause cross-iteration dependency are `minChange` and the `tour` array. There is a possibility that they both will be updated during each iteration of the texttwhile loop and if iteration `i+1` reads before iteration `i` write then there will be a read-after-write hazard, which a true dependency, that cannot be circumvented.
The `while` loop must therefore be implemented sequentially in the Futhark implementation.

Swap
Once the program has computed which `change` has the best impact on the tour (in the two opt move code section) the program proceeds to make these changes. This is done in the `swap` section of the code, lines 6-10 in the pseudo-code of the `while` loop above. To transform the sequential computation done in the `if`-statement to parallel code a `map` function is used. The `map` function

will in parallel swap all the elements between the indexes `minI` and `j` in the `tour` array to produce a tour route that has a better cost than before swapping.
In the following pseudo-code of the `swap` code the `map` function will know the indexes `minI`, `j`, the `tour` array and the `tourSize` value before executing.

```
    map(\ind ->
        if ind < minI || ind > j then
            tour[ind]
        else
            tour[j - (ind - minI)]
    ) (iota tourSize)
```

5.2.3 Nested inner `for` loops in parallel

In the section above, section 5.2.2, the outer `while` loop is discussed. Inside this loop, there is a nested `for` loop that computes the 2-opt-move by finding the `change` with the greatest impact for the current tour. The sequential implementation of the 2-opt-move is demonstrated in the following pseudo-code:
Sequential pseudo code of the nested inner `for` loops

```
... // 2 opt move
    for(int i = 0; i < (cities - 2); i++) {
        for(int j = i+2; j < cities; j++){
            change = distM[tour[i]][tour[j]] + distM[tour[i+1]][tour[j+1]] -
            (distM[tour[i]][tour[i+1]] + distM[tour[j]][tour[j+1]]);
            if(change < minChange){
                minChange = change; min_i = i; min_j = j;}}}...
```

There is a cross-iteration dependency in the nested `for` loops for the variable `minChange`. However, since this dependency is a write-after-read hazard it is possible to store each `change` into an array and then after the execution of the `for` loops use a `reduce` function to determine which `change` should be written to `minChange`. This way the `if`-statement with the cross-iteration dependency is eliminated. All the other dependencies are inside the same iteration and will not cause cross iteration dependencies.
It is therefore possible to transform the nested `for` loops to run in parallel and this is shown in the following pseudo-code:
Parallel pseudo code of the nested inner `for` loops

```
    let changeArr = map (\ind ->
                         let i = Iarr[ind]
                         let iCity = tour[i]
                         let jCity = tour[Jarr[ind] + i + 2]
                         ...
                         in   ((dist[iCity * cities + jCity] +...)
                         ) (iota totIter)
    in reduce changeComparator (maxInt, maxInt, maxInt) changeArr
```

The `changeComparator` operator has the following function

```
let changeComparator (t1 : (i32, i32, i32)) (t2: (i32, i32, i32)) :
(i32, i32, i32) =
    if t1.0 < t2.0 then t1
    else
        if t1.0 == t2.0 then
            if t1.1 < t2.1 then t1
            else
                if t1.1 == t2.1 then
                    if t1.2 < t2.2 then t1
                    else t2
                else t2
        else t2
```

where it takes two 3-tuples and compares which has the minimum value in lexical order.
This compare operator, `changeComparator`, is associative

```
changeComparator (change1, i1, j1) (changeComparator (change2, i2, j2)
    (change3, i3, j3)) =
changeComparator (changeComparator (change1, i1, j1) (change2, i2, j2))
    (change3, i3, j3)
```

and commutative

```
changeComparator (change1, i1, j1) (change2, i2, j2) =
changeComparator (change2, i2, j2) (change1, i1, j1)
```

and can therefore be used by the Futhark `reduce` function [8].

To transform the nested inner `for` loops to run in parallel they must be flattened. This is done by computing and storing the `i` values and the `j` values produced by each iteration of the outer nested `for` loop and the inner nested `for` loop, respectively.
The `i` values produced by the outer nested `for` loop are:

$$\texttt{i}_{produced}: \ [0, 1, 2, .., cities - 4, cities - 3]$$

The `j` values produced by the inner nested `for` loop are:

$$\texttt{j}_{produced}: \ [[2, 3, .., cities - 1], [3, 4, .., cities - 1], .., [cities - 2, cities - 1], [cities - 1]]$$

However two-dimensional irregular arrays are not flat-data and must therefore be flattened by the use of a shape array and a data array [8].
The flat data representation for the `i` and `j` array's must have the form

$$\texttt{i}: \ [0, 0, 0, 0, .., 0, 1, 1, 1, .., 1, 2, 2, 2.., 2, .., cities - 4, cities - 4, cities - 3]$$

$$\texttt{j}: \ [0, 1, 2, .., cities - 3, 0, 1, 2, .., cities - 4, .., 0, 1, 0]$$

Note that the values in the `j` array are different from the values in the $\texttt{j}_{produced}$ array as they are just a stepping stone to compute the correct `j` values later on in the program.
In the flat data representation, the `i` and `j` arrays must have the same size, *totIter*, which is the

total number of iterations of the nested `for` loops. Tot iter is the number of swaps that needs to be evaluated and is computed with the formula:

$$totIter = \frac{(cities - 1) \times (cities - 2}{2}$$

To compute the `i` and `j` arrays, a flag array must be computed. The flag array can be computed based on a shape array and a data array. With the computed flag array, `flagArr`, the `i` array can be computed with the following Futhark command:

```
    Iarr = scan (+) 0i32 flagArr |> map (\x -> x-1)
```

and the `j` array can be computed with the following Futhark command:

```
    Jarr = segmented_scan (+) 0i32 (map bool.i32
                                         (flagArr :> [totIter]i32)
                                    ) (replicate totIter 1i32)
                                    |> map (\x -> x-1)
```

The work inside the flatten nested inner `for` loops can be done by collecting the `i` and `j` values from the global `i` and `j` arrays to perform the calculations of `change`.

5.2.4 Random Restarts

To randomize a tour for each climber produce code to generate a random number. This code is of mediocre quality, but it is good enough to generate random indexes so that an initial tour route can be swapped randomly. Therefore this way of generating random numbers is the one used in the Futhark implementation.

5.3 Limitations of Futhark Implementation

There was a limitation to the Futhark language during the development time, where it did not allow the implementation to run on a dataset with more than about 200 cities. Due to this limitation, a CUDA implementation was created instead, see more in section 6. Since then this limitation has been removed and the Futhark implementation can now run on data sets with a large number of cities. However, there is still exists a bug in the implementation as it does not terminate when running on 1.000.000 climbers on a dataset with 200 cities or more.

6 CUDA Implementation

The sequential implementation of the 2-Opt algorithm has been analyzed and transformed into a working parallel implementation in the Futhark language, see section 5. However, there were limitations to the Futhark language when implementing the program that did not allow the program to run on data sets with more than 200 cities (this limitation no longer exists). To be able to run an implementation of the 2-Opt algorithm on more than 200 cities a parallel CUDA implementation was created. The CUDA implementation is based on the Futhark implementation. How to transform the Futhark implementation into a working implementation in CUDA, is discussed in the following sections.

This section introduces

- section 6.1 gives a short introduction to the GPU architecture and CUDA
- section 6.2 gives a short introduction for memory management
- section 6.3 describes the translation from the Futhark implementation to the CUDA implementation
- section 6.4 gives a short description of an implementation where the distance matrix is assumed to fit in shared memory.
- section 6.5 gives a high-level explanation of a way to optimize the CUDA implementation
- section 8.2 introduces some limitations of the CUDA implementation.

6.1 Short introduction to the GPU architecture and CUDA

A parallel program will have a mix of sequential parts and parallel parts and will have to run on the CPU and the GPU to maximize its performance. The CPU is designed to execute a sequence of operations, called a thread, as fast as possible. It can only execute a few threads in parallel. The CPU has therefore largely devoted its transistors to data caching and flow control, see the distribution of the CPU in figure 4. The GPU on the other hand is designed to execute thousands of threads in parallel. Each thread will have a slower performance but overall will achieve greater throughput. As the GPU is designed for highly parallel computations its transistors are largely devoted to data processing, see the distribution of the GPU in figure 4 [1].

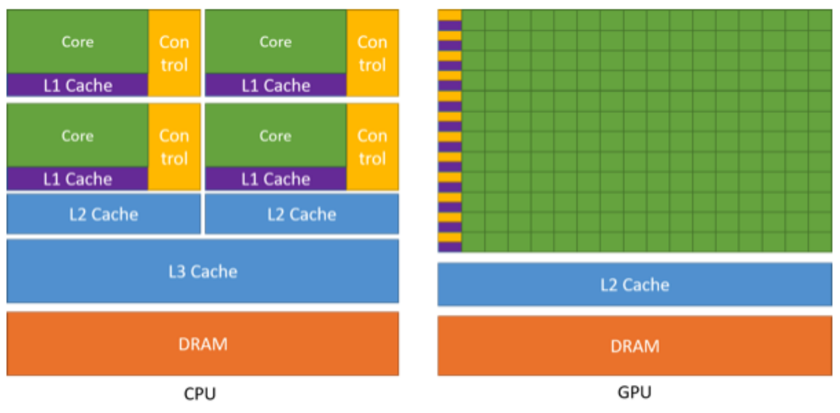


Figure 4: Distribution of transistors [2]

A way to program an implementation for the GPU is to use CUDA which is a general-purpose parallel computing platform and programming model. CUDA allows a programmer to define C++ functions, called kernels, which are invoked by the host CPU. The architecture of a NVIDIA GPU (which CUDA runs on) has a scale-able array of multi-threaded Streaming Multiprocessors (SMs). So when a kernel is called it will execute a grid of thread blocks where each thread block is distributed to an available SM. Each thread block may contain up to 1024 threads on current GPUs and a SM allows each thread of a thread block to be executed concurrently and for multiple thread blocks to be executed concurrently. Each thread and block has a unique id that is useful for when programming[1].

6.2 Memory management

The memory hierarchy of a CUDA-capable GPU is shown in figure 5

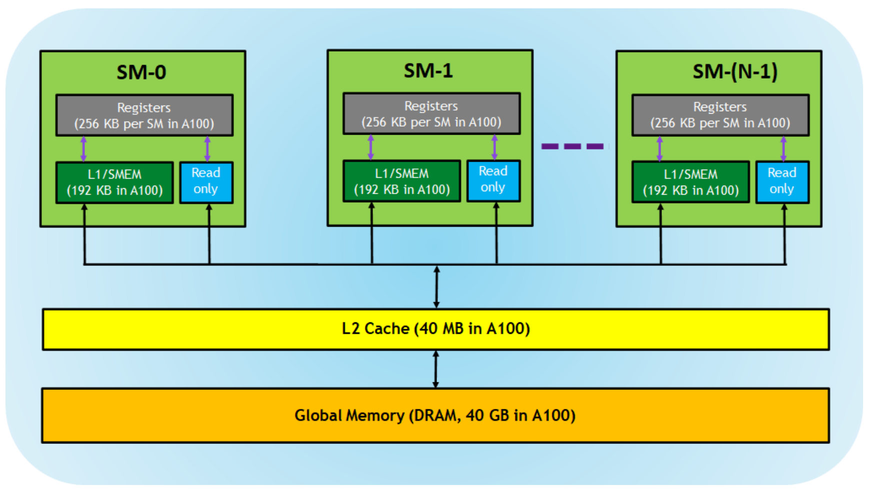


Figure 5: Memory hierarchy in GPUs [11]

All SM's have registers, L1/shared memory and read-only memory. The registers are assigned to threads individually by the compiler. These registers are private to a thread so other threads can not access them. The L1 cache/shared memory is a fast on-chip scratchpad memory. The shared memory can be shared by all the threads of a CUDA block. The read-only memory is read-only to the kernel code and can consist of an instruction cache, constant memory, texture memory, and read only (RO) cache.
The L2 cache and global memory are shared across all SM's. They are larger memories, but they are slower than the fast on-chip memory such as shared memory [10].
Global memory can be accessed both by the host (CPU) and the device (GPU). This way the CPU and GPU can transfer data between each other. The global memory can also be accessed by multiple kernels, this allows kernels to communicate with each other. Storing and loading to global memory is important for the execution of a kernel. These transactions are slow and it is desirable to perform as few transactions as possible. If consecutive threads access consecutive memory accesses, this can be done in one memory transaction and is called coalescing [12]. If the threads access nonconsecutive memory, up to 32 transactions need to be for one access. Coalescing will be used as much as possible in the CUDA implementation described in the following sections.

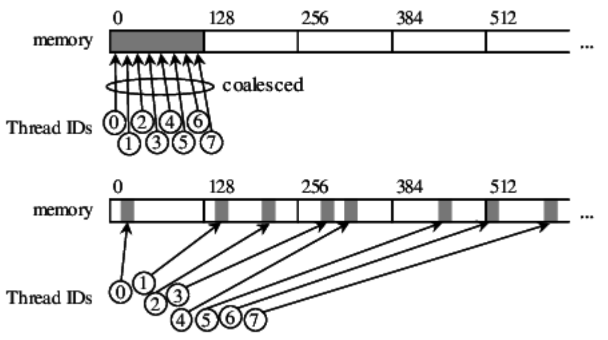


Figure 6: Coalesced and non Coalesced access depicted. Figure is from Keskin et.al paper "Real-Time FFT Computation Using GPGPU for OFDM-Based Systems"[18]

6.3 First version: The original

Three CUDA implementations are developed, all based on the Futhark implementation. The first version, which is called the original version, is the direct transformation from the Futhark implementation to the CUDA implementation. The second version, which is discussed in section 6.4, and the third version, discussed in section6.5 are optimized implementations of the first version.
This section will introduce how to transform the Futhark implementation into a CUDA implementation.
The pseudo-code for the Futhark implementation is:
Pseudo code of the Futhark implementation

```
let findCheapestCost [m] (cities : i32) (numClimbers : i64)
                    (distM : [m]i32) : i32 =
    let totIter = ((cities-1)*(cities-2))/2  --Compute total iterations
    let flagArr = flagArrayGen cities          --Compute flag array
    let Iarr = scan (+) 0i32 flagArr |> map (\x -> x-1) --Compute i array
    let Jarr = segmented_scan (+) 0i32 flagArr (replicate totIter 1i32)
                                        |> map (\x -> x-1) --Compute j array
    let allLocalBestCost = map(\climber ->
            let localTour = mkRandomTour offset cities --Compute random tour
            --Compute local best tour with 2-opt-move
            let localBestTour = twoOptAlg distM tour Iarr Jarr cities totIter
            in cost localBestTour distM --Compute cost of best tour and store it
        )(iota numClimbers)
in reduce costComparator maxInt allLocalBestCost --Collect the overall best tour
```

The number of cities (`cities`), the number of climbers (`numClimbers`), and the distance matrix (`distM`) that hold the distance between all the cities is known before the function `findCheapestCost` is executed. In the following sections, different parts of the pseudo-code for the Futhark implementation will be transformed into a CUDA implementation.

6.3.1 Initialize

Compute `i` and `j` values
In order to execute the 2-Opt algorithm in the `map` function, line 8-14 in the Futhark pseudo code, there are values that need to be initialized first. `totIter`, line 3, is the number of total iterations of the nested inner `for` loops in the sequential pseudo code that computes the 2-Opt moves. This is implemented the same way in the CUDA code as in Futhark. The flag array, `i` array, and `j` array, line 4-7, are implemented quite similarly to the Futhark implementation. Kernels have been made that do efficient implementations of data-parallel operators such as the inclusive scan operator and the inclusive segmented scan operator that are used to compute the `i` and `j` arrays. The only way the CUDA implementation of the `i` and `j` arrays diverge from the Futhark implementation is that the arrays are combined into one array with the following zip kernel:

```
__global__ void zip(int* array1, int* array2, int size){
    int glb_id = blockIdx.x * blockDim.x + threadIdx.x;
    if(glb_id < size){array1[glb_id] = (array1[glb_id] << 16) | array2[glb_id];}}
```

The `i` value is stored in the first 16 bits of an integer and the corresponding `j` value is stored in the last 16 bits of an integer. This is done to minimize the number of reads from global memory.

Compute block size
Another value that is initialized before the program executes the 2-Opt algorithm is the size of the CUDA blocks. This is computed by the host. The following code snippet is the implementation of determining the block size.

```
... if(totIter > 512){block_size = 1024;}
    else if(totIter > 256){block_size = 512;}
    else if(totIter > 128){block_size = 256;}
    else if(totIter > 64){block_size = 128;}
    else if(totIter > 32){block_size = 64;}
    else{block_size = 32;} ...
```

The size is based on `totIter`. It must maximum be 1024, as CUDA blocks currently support up to 1024 threads per block, and it must be divisible by 2, as it is used to determine the number of iterations in the `reduce` functions.

6.3.2 `map` function

There are multiple options of how to implement the `map` function, line 8-13 of the Futhark pseudo code, where the 2-Opt algorithm is executed. The Futhark program is implemented by executing a climber per thread. However, to achieve a higher degree of parallelism the CUDA implementation executes a climber per block. This way multiple threads can compute the work of the 2-Opt algorithm in parallel inside a block, exploiting the inner parallelism. This way both degrees of parallelism are exploited. The outer parallelism is exploited by assigning climbers on grid level, and the inner by using `block_size` threads to evaluate the 2-opt moves.

6.3.3 Random tour

The random tour in line 9 of the Futhark pseudo code above is computed in the `map` function by each climber. In the CUDA implementation, the random tours are computed and stored into a

tourMatrix on global memory before executing the `map` function. Each tour has the size `cities+1`, as a tour starts and ends in the same city, and the matrix has the size `cities+1` × `numClimbers`. The reason to compute and store the randomized tours before the `map` function is to exploit a higher degree of parallelism.
Following is the pseudo code for the randomized tour kernel:

```
1  __global__ void createToursColumnWise(tourMatrix, cities, climbers){
2      glo_id = threadIdx.x + blockIdx.x * blockDim.x;
3      if(glo_id < climbers){
4          for(int i = 0; i < cities; i++){tourMatrix[climbers * i + glo_id] = i;}
5          tourMatrix[climbers * cities + glo_id] = 0; //same start and end city
6          rand = glo_id + blockIdx.x; //random seed
7          for(int i = 1; i < cities; i++){ // randomize tour
8              rand = (MULT * rand + ADD) & MASK; to = rand % cities;
9              temp = tourMatrix[climbers * i + glo_id];
10             tourMatrix[climbers * i + glo_id] = tourMatrix[climbers * to + glo_id];
11             tourMatrix[climbers * to + glo_id] = temp;}}}
```

The kernel will allocate each thread to compute a randomized tour for each climber. Each thread will create an initial tour and write column-wise to the tourMatrix, lines 4-5, securing coalesced storing to global memory. Then the tour will be randomized, by each thread, with a simple randomization algorithm that creates random indexes and swaps these elements, lines 6-11.
Once the kernel has computed the randomized tours the host will call a transpose kernel, to transpose the tourMatrix. This is done so that each climber in the `map` function will achieve coalesced access when loading its randomized tour from global memory.

6.3.4 The 2-opt-move kernel

The local best tour computed for each climber uses the 2-opt-move algorithm. In the Futhark pseudo code, these computations are done by the function called `twoOptAlg`, line 11. In the CUDA implementation, these computations are done by a kernel called `twoOptKer`. The kernel will run a climber per block and in every block, the threads will work together to compute the best tour for the climber.

Shared memory
To optimize memory management, the randomized `tour` array, the `minChange` variable, and an array to store the best changes found by each thread, `tempRes`, will be stored in shared memory. The shared memory is set inside the kernel with the following code:

```
... extern __shared__ unsigned char totShared[];
    volatile ChangeTuple* tempRes = (volatile ChangeTuple*)&totShared;
    volatile ChangeTuple* minChange = tempRes + block_size;
    volatile unsigned short* tour =
                (volatile unsigned short*)(minChange + 1); ...
```

Before the kernel is called, the host will determine the size of the shared memory with the following code:

```
... size_t sharedMemSize = (cities+1) * sizeof(unsigned short) + block_size *
                      sizeof(ChangeTuple) +  sizeof(ChangeTuple); ...
```

Once the shared memory is set up, the randomized tour for this climber is read from the tourMatrix in global memory to shared memory. This is done in the following code snippet:

```
...  for(int t = idx; t < cities+1; t += block_size){
        tour[t] = glo_tours[blockIdx.x * (cities+1) + t];}
```

Note that the kernel reads a chunk with the size `block_size` (if `cities+1` is larger than `block_size`) from global memory to ensure threads have coalesced memory accesses. Whenever it is possible to read and write from the global memory in chunks of `block_size`this technique is used for this kernel.

Sequential `while` loop
The program proceeds to initialize variables needed in the sequential `while` loop of the program. This `while` loop is discussed further in section 5.2.2. The sequential `while` loop is implemented with barriers to ensure safe reads and writes to memory. These barriers are provided by the CUDA environment and are called `__syncthreads()`.

Flattened nested `for` loops
The flattened nested `for` loops inside the sequential `while` loop, discussed further in section 5.2.3, are implemented similar to the Futhark implementation.
The following pseudo-code is the CUDA implementation of the flattened nested `for` loops:

```
1  ... for(int ind = idx; ind < totIter; ind += block_size){
2          int num = glo_is[ind]; i = num >> 16;
3          j = (num & 0xffff) + i + 2; ip1 = i+1; jp1 = j+1;
4          change =
5             glo_dist[tour[i]*cities+tour[j]] + glo_dist[tour[ip1]*cities+tour[jp1]] -
6             (glo_dist[tour[i]*cities+tour[ip1]] + glo_dist[tour[j]*cities+tour[jp1]]);
7          ChangeTuple currentChange = ChangeTuple(change, i, j);
8          localMinChange = minInd::apply(localMinChange,currentChange);}  ...
```

A difference between the Futhark and CUDA implementation is how of the `i` and `j` elements are accessed. In the Futhark implementation these are stored in two arrays, which causes two reads from global memory per i and `j` pair. In the CUDA implementation, this is optimized by storing the i and `j` elements into one array, `glo_is`, causing only one read to global memory per i and `j` element. Another difference between the parallel implementations is that the Futhark implementation will compute and store `change` into an array that has size `totIter` and then reduce it. In the CUDA implementation, `block_size` number of threads will compute a new `change` where each thread might do several evaluations, if `totIter` is greater than `block_size`. For each iteration, the thread must check if the `change` computed is better than the previous iteration, line 7-8 in the pseudo-code above.

Reduce
After the execution of the flattened nested `for` loop each thread will write its best computed `change` to shared memory in the array `tempRes`. Then a `reduce` function is used to find the overall best local `change`. The following code snippet is of the `reduce` device kernel:

```
__device__ void reduceLocalMinChange(int block_size, volatile ChangeTuple* tempRes){
    int idx = threadIdx.x;
    for (int size = block_size >> 1; size > 0; size >>= 1 ){
        if(idx < size){
            tempRes[idx] = minInd::apply(tempRes[idx],tempRes[idx + size]);}
        __syncthreads();}}
```

`minInd` corresponds to the Futhark operator `changeComparator`, see more in section 5.2.3. This `reduce` function compares the elements by having half of the threads in a block (in the first iteration) compare two elements of `tempRes` and write back the best one to `tempRes`. Next iteration $\frac{1}{4}$ of the threads will compare two elements and write back the best one. This will continue until every element has been compared and the best is stored in `tempRes[0]`. Note that `tempRes` has to have the same size as `block_size`, so if there are not enough elements to fill out `tempRes`, it must be padded with neutral elements.
All other `reduce` device kernels in the `twoOptKer` kernel reduces in the same way.

Swap
When the overall best local `change` is found the tour can be swapped and the CUDA implementation of swapping is very similar to the Futhark implementation. See Futharks implementation in section 5.2.2. There is no data dependencies since only $\frac{CitiesToChange}{2}$ threads are used to reverse the elements, and only that specific thread accesses the same memory location.

Cost
Once the best tour for a climber is computed the cost of that tour must be determined. This is done with a sum `reduce` function similar to the Futhark implementation of a sum `reduce` function. See the Futhark implementation in section 5.2.1.

6.3.5 Outer Reduce

The final `reduce` function, line 14 of the Futhark pseudo-code at the beginning of section 6.3, finds the tour with the best cost. This is a simple `reduce` compare function that determines the minimum of two values.
The way this `reduce` function is implemented in CUDA is by reducing across multiple blocks. Each block will reduce on `block_size` $\times 2$ elements so that all the threads will compare two elements during the first iteration. This way there are no idle threads during the first iteration as there are with the other `reduce` functions implemented in the CUDA implementation.
This implementation is inspired by NVIDIA presentation on Optimizing Parallel Reduction in CUDA [13].

6.4 Second version: The 100 cities

The second version is called the 100 cities version. It is more limited than the original version as it can only run datasets with up to 100 cities, because of the limited size of shared memory. This version is created to check the potential speedup and can be compared with the solution made by to O'Niel et. al's. Both the O'Niel et. al solution and the 100 cities version store the distance matrix (`distM`) into shared memory for quicker memory access. The distance matrix holds the distance between all the cities. Since the shared memory capacity is limited the information from a dataset with more than 100 cities can not fit into it. Even though the 100 cities version showed some speed-up compared to the original version, the speed-up was not big enough to have the limitation of only running smaller datasets.

6.5 Third version: The calculated

The third version is called the calculated version. It is very similar to the original version, however, the main difference is the way it computes the `i` and `j` values. In the original version the `i` and `j` values are computed during initialization where they are written to global memory, see more in section 6.3.1. They are then used in the flattened nested loops where the values are read from global memory, see more in section 6.3.4, subsection "Flattened nested `for` loops". The writes and reads to global memory are expensive and therefore it would be desirable to eliminate these accesses by having each thread compute the `i` and `j` values when they need them during the flattened nested loops, instead of reading them.
By analyzing the patterns of how the `i` and `j` values are produced in each iteration of the sequential nested `for` loops, see the pseudo-code of the sequential nested `for` loops below:

```
for(int i = 0; i < (cities - 2); i++) {
    for(int j = i+2; j < cities; j++){...}}
```

it was possible to construct a mathematical expression that can compute the correct `i` and `j` values for a given thread. This is explained in depth in appendix A.
Following is a summary of the mathematical expression created:
The formula to compute the value `i` by a thread with a threadID:

$$n_{curr} = \lfloor \frac{-1 + \sqrt{1 - (4 \times 1 \times (-(2 \times (totIter - threadId))))}}{2 \times 1} + 0.9999 \rfloor$$

$$\texttt{i} = (cities - 2) - n_{curr}$$

The formula to compute the value `j` by a thread a threadID:

$$\texttt{j} = \texttt{i} + 2 + (threadId - (totIter - \frac{n_{curr} \times (n_{curr} + 1)}{2}))$$

These formulas are inserted into the program of the calculated version and can be seen in the following code snippet

```
1  ... for(int ind = idx; ind < totIter; ind += block_size){
2          d = 1-(4*(-2*(totIter-ind)));
3          tmp = ((-1+(sqrt((float) d)))/2)+0.9999;
4          curr = (int) tmp;
5          i = (cities-2) - curr;
6          j = (i+2) + (ind-(totIter-((curr*(curr+1))/2)));
7          ... } ...
```

Line 2 computes the discriminant of the quadratic formula used to compute the value `i`. Line 3 computes the quadratic formula and adds 0.9999. Line 4 takes the floor and now n_{curr} is computed. Line 5 computes the value `i`. Line 6 computes the value `j`.
.

7 Evaluation

To evaluate our solution it was tested on different public data sets from the TSPLIB95 library [15] with varying numbers of cities.
All the data sets are represented as complete graphs, and the data is either in the form of Euclidean 2D points, or pre-constructed matrices.
The data sets which have been tested, have known optimal solutions, which makes it easy to calculate how accurate our solution is.

This section starts by comparing our sequential versions, of the 2-Opt local search, Ant Colony Optimization, and Simulated Annealing algorithms.
Since the 2-Opt version was chosen to be implemented in CUDA, the accuracy of the solutions is evaluated depending on the number of climbers. It is measured in how close to the optimal solution, the solutions generated are.

After this, we will see compare our solution with the publicly available solution made by Molly A. O'Neil, Dan Tamir, and Martin Buscher, and described in the paper "A parallel GPU version of the Traveling Salesman Problem" [21] and will from now on be referenced as [O'Neil et al.].

The section ends with an evaluation of how well our solution uses the hardware GPU, by measuring the throughput in GB/s. This approach is chosen since our solution is memory bound, i.e. each operation has low arithmetic density. It is then evaluated on how well it uses the hardware GPU based on the big memory throughput of the GPU

To be able to repeat the tests and for testing the correctness of the solution, we have chosen a specific seed for the random number generator. Repeated testing would give the same result, even though the algorithm should be randomized. To revert this change seed for the random number generator should be changed to be some sort of randomized number (such as the time of day).

7.1 Hardware description

The GPU used for all testing is an NVIDIA A100 PCIe 40 GB [28]. The GPU has the following relevant specifications:

- Release year: 2020
- Memory Size: 40 GB
- Top Bandwith: 1555 GB/s
- SM Count: 108
- L1 Cache: 192 KB
- L2 Cache: 40 MB
- Shared mem size = 48 KB

7.2 Data sets

All the data sets used for testing our solution are from the TSPLIB95 library. The TSPLIB95 is a library consisting of TSP instances and instances of problems related to TSP [26]. It is publicly available and has known optimal solutions for most data sets. The data set and the optimal known costs will be used to evaluate our solution and make it possible to compare with other solutions that support these data sets, such as the solution by [O'Neil et al.].

Only TSP data sets are used from the library. Also, the solution only works with specific data sets. These data sets are then ones that use the "EUC_2D" weight type, which denotes Euclidean distances in 2 dimensions, Or the MATRIX weight type, where all distances are pre-computed in a distance matrix.
Below d a table with all the data sets used to evaluate our solution is provided. The table denotes how many cities there are in the data set and the optimal known cost for the data set.

Data set	nr. cities	optimal cost
swiss42.tsp	42	1273
berlin52.tsp	52	7542
kroA100.tsp	100	21282
kroB100.tsp	100	22141
kroA200.tsp	200	29368
pr439.tsp	439	107217

Table 1: TSP data sets used for evaluation

7.3 Comparisons of sequential versions

To compare the sequential versions, implementations in c were made for each of them. The parameters for Ant Colony Optimization and Simulated Annealing are predetermined, but for the best results, optimal parameters should be found for the specific data set al.l the sequential versions were tested on the swiss42.tsp data set, since there appeared to be some memory issues for the SA and ACO implementations, on data sets with a high number of cities.
All sequential versions report the quality of their solution, after a specific number of iterations.
For the 2-Opt implementation a solution is reported every time a local optimum is found and reports how many 2-opt changes are evaluated, no parameters were needed for this algorithm.
SA annealing reports the solution after 10 neighboring solutions have been evaluated. The starting temperature was chosen to be $T_0 = edges^3$, and the cooling schedule was a multiplicative $T = T_0 \cdot \alpha^{iter}$ ACO reports the solution every 10 iterations. Since the ACO algorithm gets a specific number of iterations, this has been chosen to be 1000. The parameters for ACO were

$$\begin{aligned} Q &= 100.0 \\ alpha &= 1.0 \\ beta &= 5.0 \\ rho &= 0.5 \end{aligned}$$

based on findings from Wei [32].

The tests gave the following results. The cost of the solution is depicted on the Y axis and the number of iterations is on the X-axis:

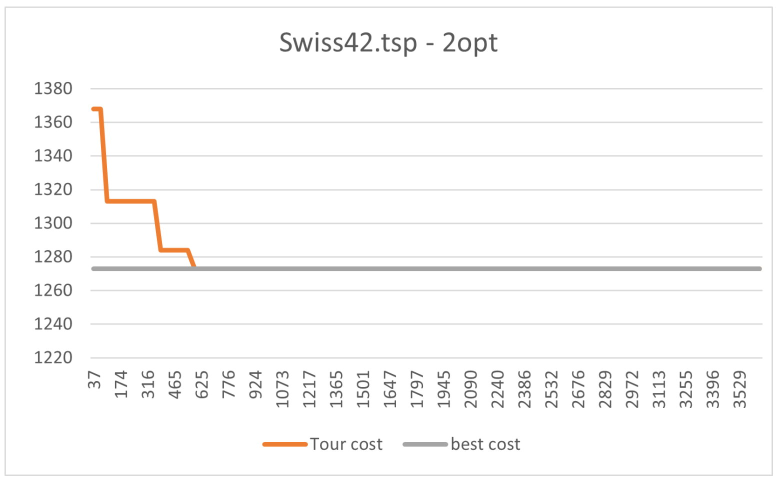


Figure 7: Convergence of quality as a function of iterations for 2-opt sequential algorithm

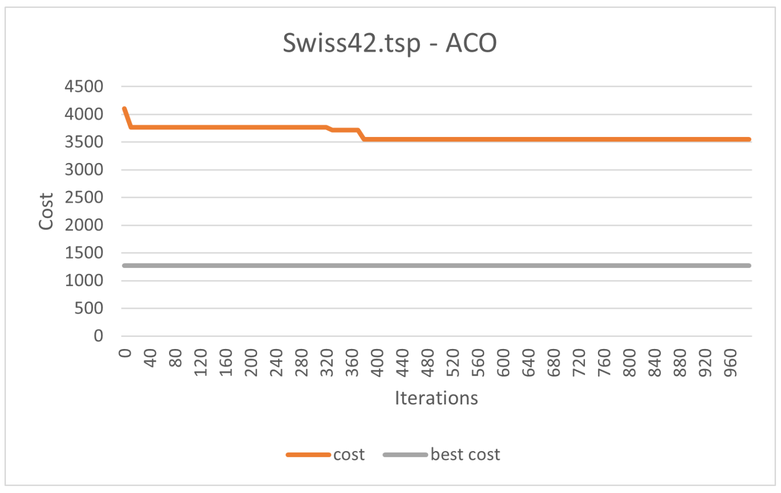


Figure 8: Convergence of quality as a function of iterations for ACO sequential algorithm

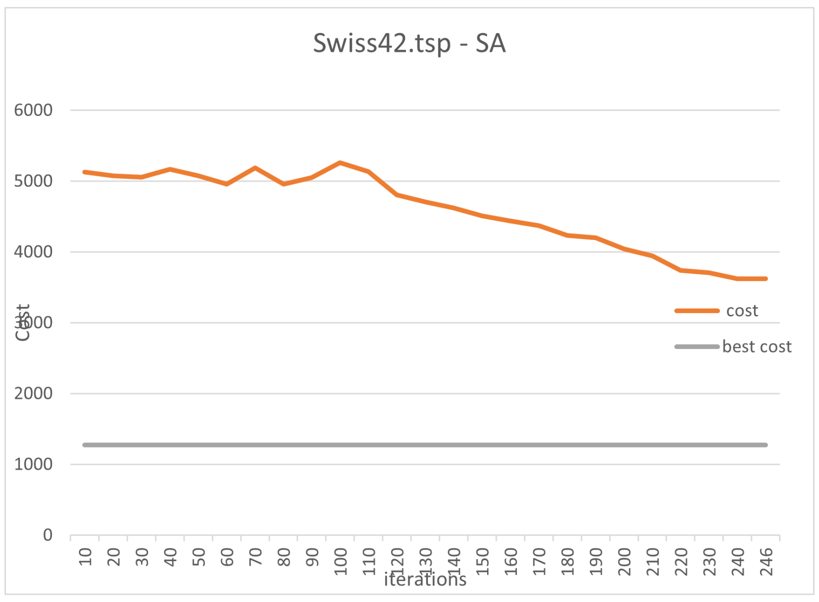


Figure 9: Convergence of quality as a function of iterations for SA sequential algorithm

From this, it can see that the 2-Opt algorithm quickly finds a local optimum, and finds the best quality solution after 625 2-opt evaluations. It should be noted that iterations are each 2-opt considered, not only those taken.
The ACO solutions start with a very bad solution and do not converge quickly. After 400 iterations, no improvements seem to be found. It should be noted, that ACO is likely to perform much better if optimal parameters are determined.
The SA algorithm looks like it starts by exploring the global search space by accepting changes that worsen the quality of the solution, and ends by slowly converging to a better solution. However, it stops before getting close to the global optimum. The SA algorithm terminates when the temperature is too low, indicating that the cooling schedule might cool too quickly since only 246 iterations were done. This might get better results with a better choice for cooling schedule, cooling factor, and/or start temperature.

Based on these tests, the 2-Opt algorithm shows the best promise for finding good-quality solutions, for TSP.

7.4 Accuracy

To test the accuracy of our solution, the cost of the solution was evaluated on the number of climbers. The cost is then compared to the known optimal cost 1. To have an acceptable solution, we will define it as a solution that deviates less than 5% from the optimal cost. The accuracy tests were done for three different data sets:

- berlin52.tsp: a small data set consisting of data points from 52 different cities
- KroA100.tsp: a medium data set consisting of 100 cities

- kroA200.tsp: a big data set consisting of 200 cities

The tests were done with randomized seeds, and each every time they reported very similar results, with some minor deviation.

7.4.1 Accuracy evaluation of Berlin25.tsp

Berlin25.tsp data set was evaluated on a range of 1-200 climbers. After this, a solution with optimal cost was found.

Climber	Lowest cost computed	Accuracy (%)
1	8401	89.7750268
2	8401	89.7750268
5	8118	92.9046563
10	7713	97.7829638
20	7542	100
50	7542	100
100	7542	100
200	7542	100

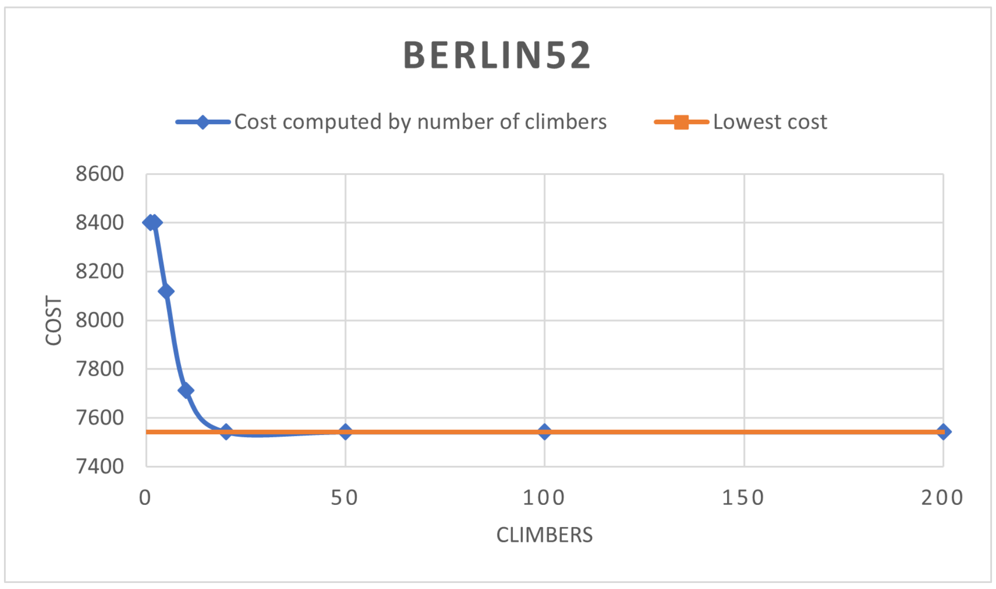


Figure 10: accuarcy evaluated as a function of climbers for the ber52 data set

A search with only one climber resulted in a solution where the cost deviated ca. 10% from the optimal solution. This shows that the sub-optimal solution found by only one climber was not close enough to our acceptance rate, but not too far off either.

The next interesting number of climbers is 10, as a solution with 97% accuracy was found. This shows that the number of climbers needed to reach an acceptable solution is quite small, for this data set.

A solution that yields the optimal cost, is found at 20 climbers. After this, the cost does no longer change. This shows that is only necessary to spawn 20 blocks for the GPU, to solve this data set. This indicates that these small problems do not use the GPU fully. This will be explained in greater detail in the ”Measuring throughput” section 7.7.1.

7.4.2 Accuracy evaluation of kroA100.tsp

Climber	Lowest cost computed	Accuracy
1	22805	93.32164
10	22236	95.70966
100	21601	98.5232165
1000	21450	99.2167832
10000	21292	99.953034
20000	21292	99.953034
50000	21292	99.953034
75000	21282	100
100000	21282	100
125000	21282	100

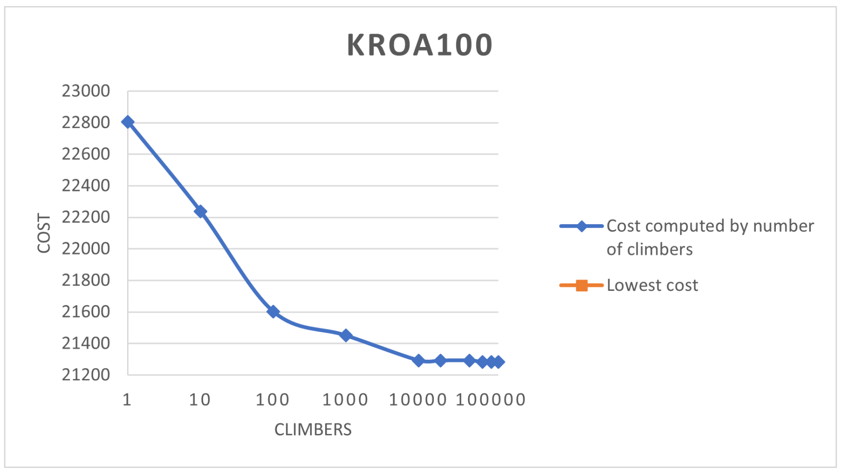


Figure 11: accuarcy evaluated as a function of climbers for the kroA100 data set

The kroA100.tsp data set was tested with a bigger range of 1 - 125000 climbers, as this bigger data set required a greater number of climbers to find the optimal solution.

However, the sub-optimal solutions found are in general quite close to the optimal solution as we can see by the solution obtained by 1 climber, which gets a 93% accuracy. This increases steadily and we get within our acceptance rate on 1000 climbers where we have a 99.21% accuracy.
From here the increase to 10.000 climbers gives 99.9 % accuracy and this does not change until the number of climbers increases to 75.000 where the optimal solution is found together with the optimal cost.

This shows that we can obtain an acceptable solution with a relatively small number of climbers.

7.5 Accuracy evaluation of kroA200.tsp

For the ”big” kroA200.tsp data set we tested from 1, 100, 1000 - 10.000.000 climbers.

Climber	**Lowest cost computed**	**Accuracy**
1	46062	63.7575442
100	31119	94.3732125
1000	30346	96.77717
10000	30200	97.2450331
50000	29974	97.9782478
100000	29974	97.9782478
500000	29850	98.3852596
1000000	29850	98.3852596
1500000	29689	98.9187915
5000000	29665	98.9988202
10000000	29665	98.9988202

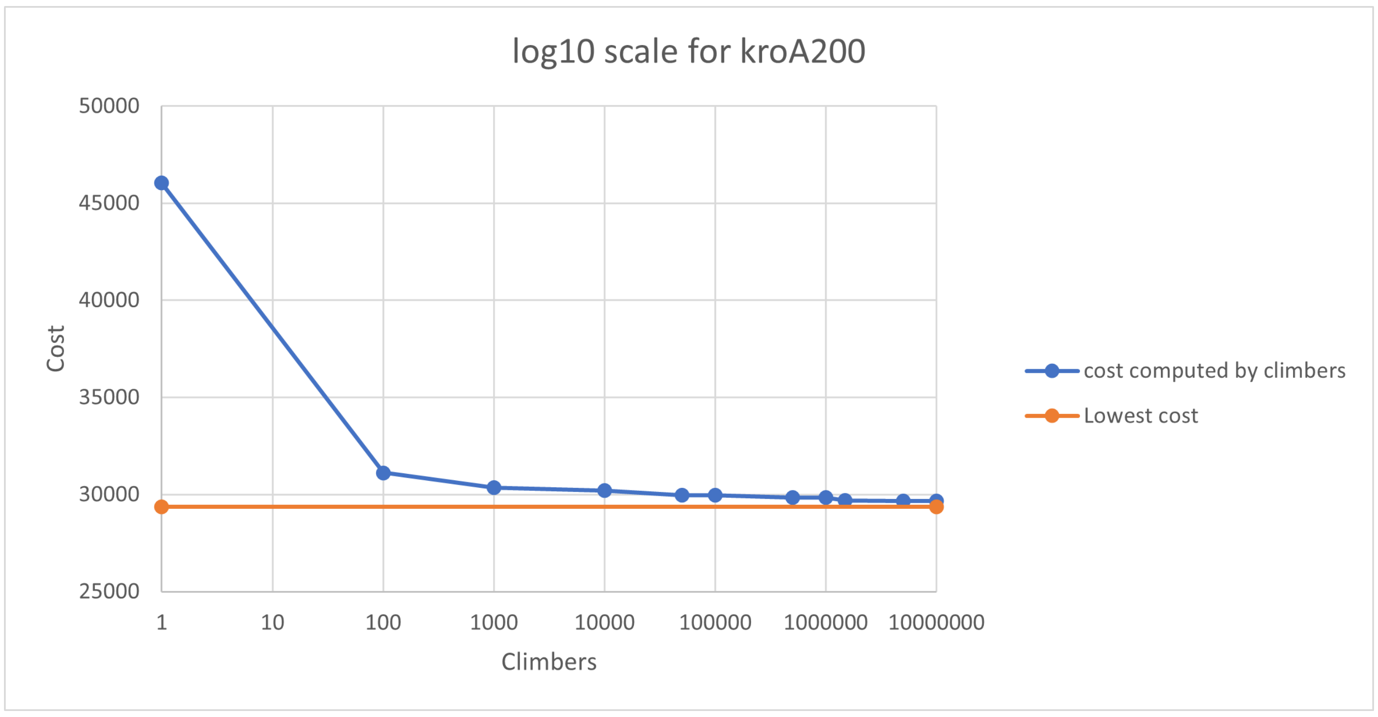


Figure 12: accuracy evaluated as a function of climbers for the kroA200 data set

for this data set, 97% accuracy is reached after 10.000 climbers and only increases marginally from here. The optimal solution was not obtained with our solution on this data set. After 10.000.000 climbers we achieved an accuracy of 98.99 % percent, so a solution within the acceptance range was found. If a lot of different seeds were used, the optimal solution might be found

7.6 Comparison with [O'Neil et al.]

We compare our solution with the [O'Neil et al.] [21] solution since both use the same algorithm, but different approaches. The two solutions were compared with respect to the run time since both implementations find a solution within the acceptance rate. Both solutions are tested on the same GPU, NVIDIA A100 PCIe 40 GB 7.1. The TSP data sets which were tested were kroA100.tsp and Berlin52.tsp The reason only 2 data sets are used for testing, is because of the limitations of the solution by [O'Neil et al.]. The solution by [O'Neil et al.] can only handle data sets with ≤ 100 cities since the distance matrix must fit in shared memory. Furthermore, the solution by [O'Neil et al.] requires a specific "weight_type" of their data sets (EUC2D), so data sets with explicit sizes cannot be used.

7.6.1 run time

We have measured the run time in milliseconds (ms) for both our solution and [O'Neil et al.] solution.

The solution measured the run time of the three different versions of our solution:

- The original version, where both I and J arrays and the distance matrix reside in global memory.
- 100cities where the distance matrix and I and J array fit in shared memory, such that temporal locality is better used since the distance matrix is accessed frequently the shared memory has lower latency than global (
- and the calculated I and J where we calculate the values for I and J instead of keeping them in global memory.

The result of the tests were:

Table 2: Test results for the kroA100.tsp data sets

Time in milisecs	300.000 climbers	200.000 climbers	100.000 climbers	50.000 climbers	10.000 climbers	1.000 climbers
TSP - Texas	546	337	144	144	143	136
TSP - Original	1187	792	396	198	40	5
TSP - 100 cities	1070	714	357	179	36	4
TSP - calculated i and j	1150	765	382	191	39	5

Table 3: Test results for the ber52.tsp data set

Time in milisecs	100.000 climbers	50.000 climbers	10.000 climbers	1.000 climbers
TSP - Texas	18	17	17	17
TSP - Original	117	59	12	2
TSP - 100 cities	108	54	11	2
TSP - calculated i and j	118	59	12	2

From our test run of the data sets, the following can be seen. Our solution are faster for smaller numbers of climbers, while the [O'Neil et al.] solution is faster for a greater number of climbers. This can be explained by the design of our solution as mentioned in 6.3.2. The [O'Neil et al.] solution only uses the outer level of parallelism, resulting in no need for communication between threads, except for comparisons of results. This results in the [O'Neil et al.] performing better when many blocks are spawned, as no minimal communication between threads is necessary. . This however also means that their solution needs a lot of climbers to satisfy the GPU. Instead, our solution maps one climber to one block, increasing the degree of parallelism up to a factor of 1024 (maximum block size). This however requires communication, and synchronization between threads is required, which results in some overhead run time.

7.6.2 Discussion of results

The comparison of the solution will start by evaluating the results for 1.000 climbers and 10.000 climbers on the kroA100.tsp data set. Here our solution runs $136/5 \approx 27.2$ faster than the solution by [O'Neil et al.] and even on 10.000 climbers, we are $143/40 \approx 3.5$ times faster. However, on the bigger data sets the [O'Neil et al.] solution is about $\frac{1150}{546} \approx 2.1$ times faster.

However, two things should be considered. Firstly the accuracy of the solution is very good already at 1000 climbers. As discussed in the previous section, the accuracy for this at 1000 climbers data set is within $< 0.1\%$ of the optimal solution. Meaning an acceptable solution is found very fast for 1000 climbers.

Secondly, the [O'Neil et al.] solution also seems to have an undiscovered bug. When testing with a very high number of climbers, the [O'Neil et al.] solution reports a path whose cost is less than the known optimal solution. Costs marked with red in the result tables denote a cost less than the optimal known cost.

7.6.3 Speedup of optimizations

The speedup of our optimizations is observed here. The shared memory optimization gives a speedup of $\frac{792}{714} \approx 1.11$ which means it is 11% faster than the original version. This speedup is not enough to warrant the trade-off of not handling data sets with more than 100 cities.

The calculated i and j versions result in a $\frac{792}{765} \approx 1.03$ so 3% speedup. It is not a big speed-up but introduces no limitations, therefore this optimization will be kept.
Both of these optimizations are memory optimizations, and since speedup is achieved, it indicates that our solution is memory bound.

7.7 Performance on GPU

This section reports the bandwidth of our solution (GB/s) and compares it with the big bandwidth of the hardware, in order to demonstrate how well the GPU hardware is utilized.
To evaluate this, we measure how much throughput our solution achieves for different data sets, and compare it with the big throughput of the hardware GPU. The GPU (NVIDIA A100 PCIe 40GB) has a maximum throughput of 1.555 GB/S [28].

7.7.1 Measuring throughput

To measure the throughput of our solution, the number of global memory accesses is analyzed and multiplied by how many bytes are accessed. This results in the number of bytes handled by the GPU, and0 it then divided by the running time. The time is measured in microseconds. The result is then converted to GB/s by:

$$\frac{\#byte \cdot 1.0e^{-9} Gigabyte/byte}{\#microseconds \cdot 1.0e^{-6} second/microsecond}$$

Which can be simplified to

$$\frac{\#byte \cdot 1.0e^{-3}}{\#microseconds}$$

Only the work of the 2 opt algorithm is measured, since this is the work we want to evaluate. Profiling our solution shows that $> 98\%$ of the running time is used on the kernel responsible for the algorithm. The kernel that is measured is the "calculated " version 6.5 and in the kernel, there are the following global memory accesses.

1. When copying the initial tour from global memory to shared memory"
2. accessing distances from the distance matrix, when computing the change of a swap
3. calculating the cost of a tour created by the climber
4. copying the tour back to global memory

The number of bytes accessed is annotated in the code and can be seen in appendix B

Copying the tour from and back to global memory only happens once in the algorithm and only scales with the number of cities in the tour. Computing the cost of the tour only happens once, and scales on the number of cities as well. Accessing the distance matrix however scales with two factors, how many cities are in the tour and how many while loop iterations are needed before a local optimum is found. The first factor is easily computed, as this is just how many "iterations" there are in the flattened array multiplied by how many memory accesses are done. However, the second factor is indeterminable, since it is a convergence loop and changes from the data set to the data set.
Therefore a new kernel is created to count the number of while iterations necessary for each block, and then sum up all the while iterations with an efficient reduce kernel.

7.7.2 Results

The throughput was tested for six different data sets

- swiss42.tsp : 42 cities
- berlin52.tsp : 52 cities
- kroA100.tsp : 100 cities
- kroB100.tsp : 100 cities

- kroA200.tsp : 200 cities
- pr439.tsp : 439 cities

All tests are done over 100 tests on the GPU, where the average time was taken. The throughput has been plotted as a function of the number of climbers for the data sets. Each data set have been evaluated for 1-1000, 1-5000, and 1-10000 climbers. All the graphs depict the climbers on the X-axis and the throughput on the Y-axis. The orange line represents the big throughput of the hardware GPU.

7.7.3 swiss42.tsp

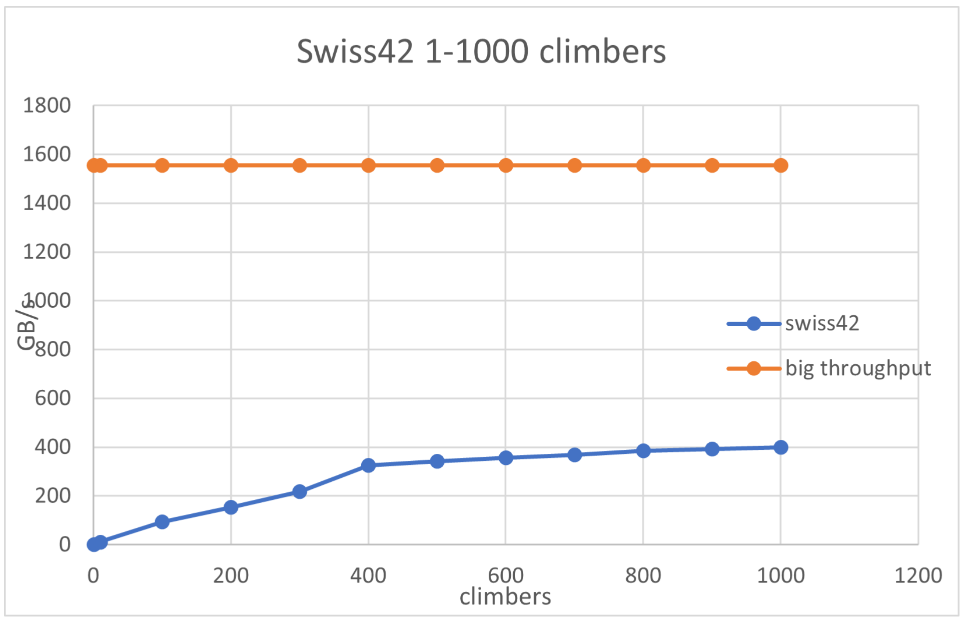


Figure 13: Throughput of swiss42 as function of climbers for 1-1000 climbers

7.7.4 Throughput on berlin52.tsp

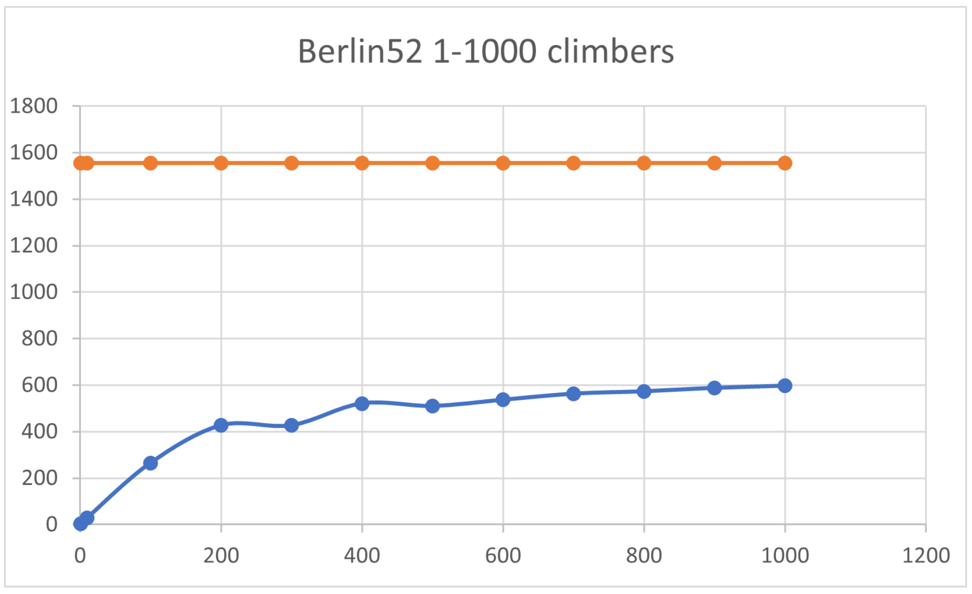


Figure 14: Throughput of ber52 as a function of climbers for 1-1000 climbers

7.7.5 Throughput on kroA100.tsp

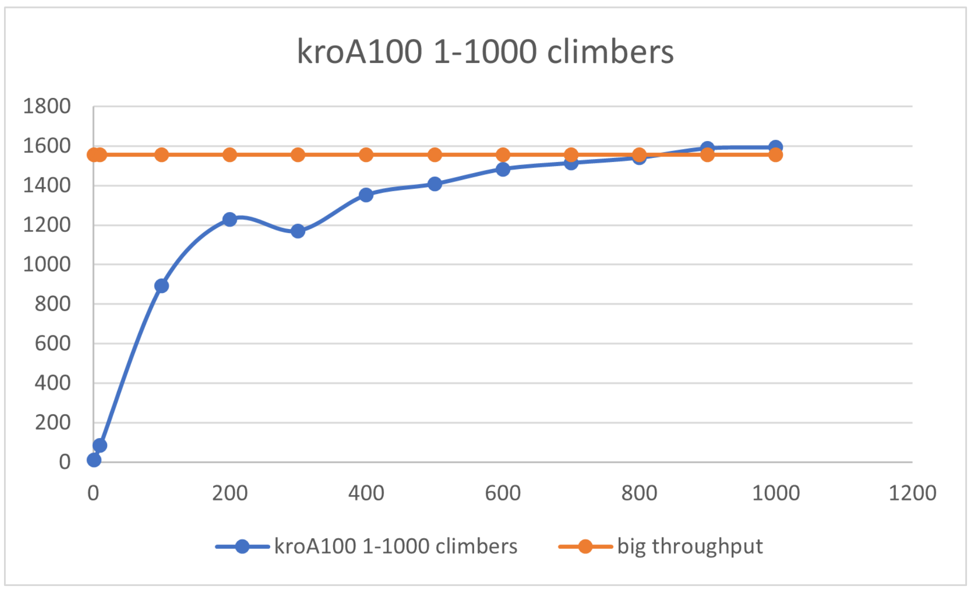


Figure 15: Throughput of kroA100 as a function of climbers for 1-1000 climbers

7.7.6 Throughput on kroB100.tsp

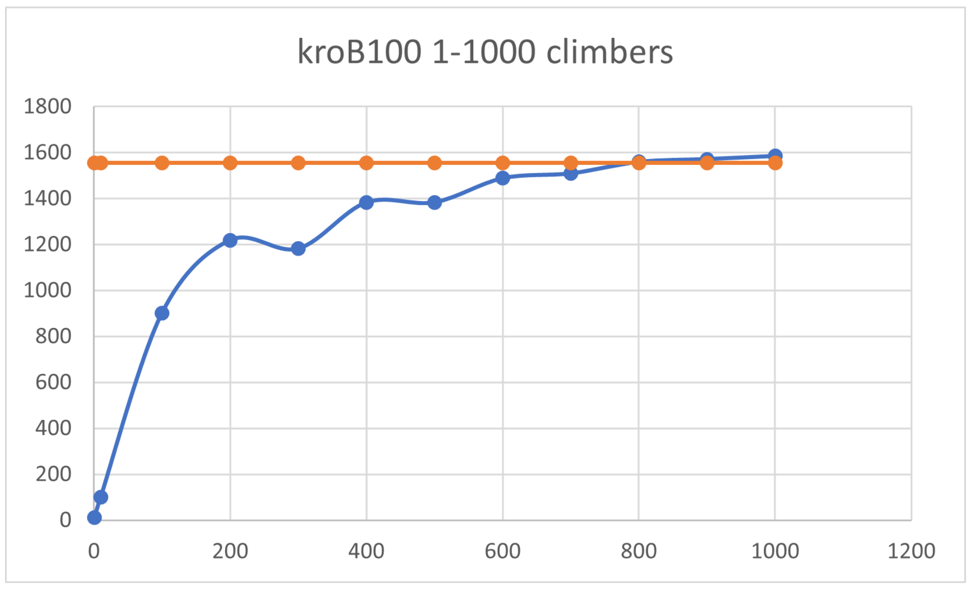


Figure 16: Throughput of kroB100 as a function of climbers for 1-1000 climbers

7.7.7 Throughput on kroA200.tsp

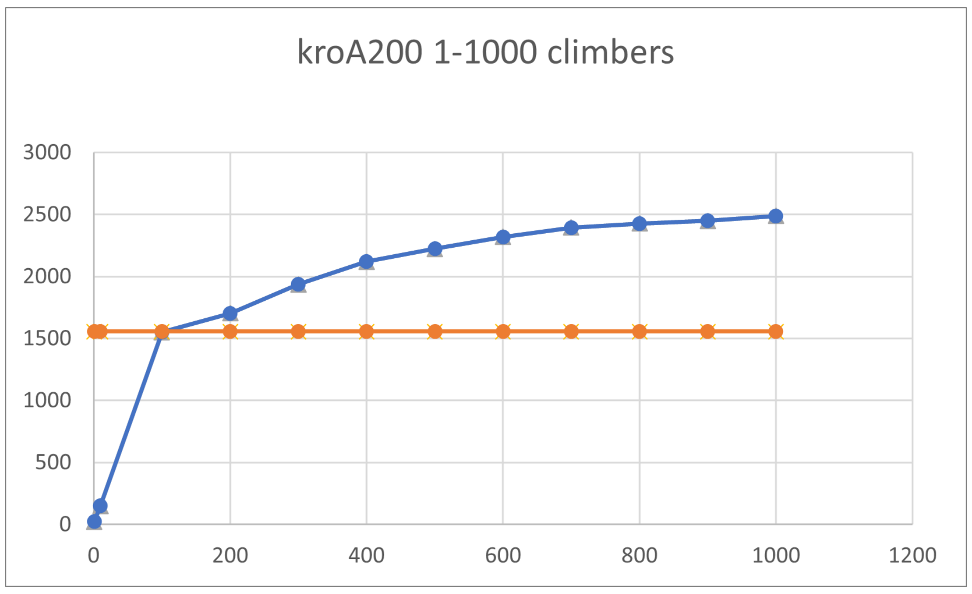


Figure 17: Throughput of kroA200 as a function of climbers for 1-1000 climbers

7.7.8 Throughput on pr439.tsp

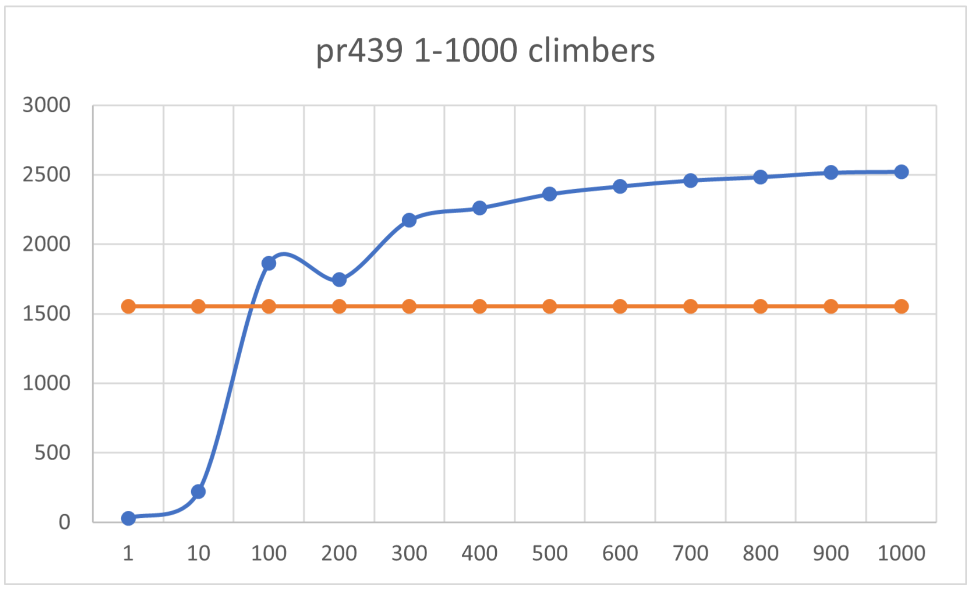


Figure 18: Throughput of pr439 as a function of climbers for 1-1000 climbers

7.7.9 Throughput on all data sets

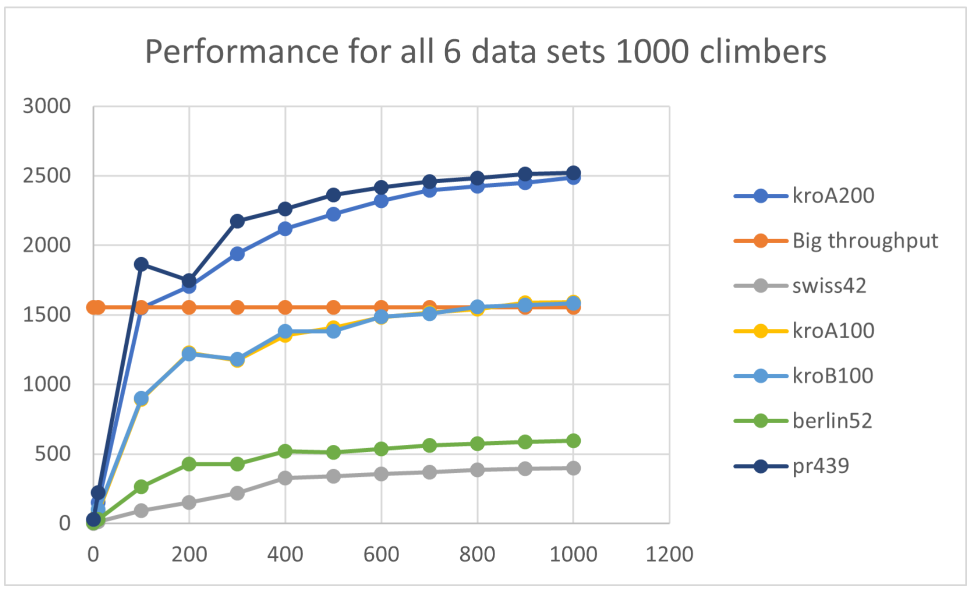


Figure 19: Throughput of all datasets for 1-1000 climbers

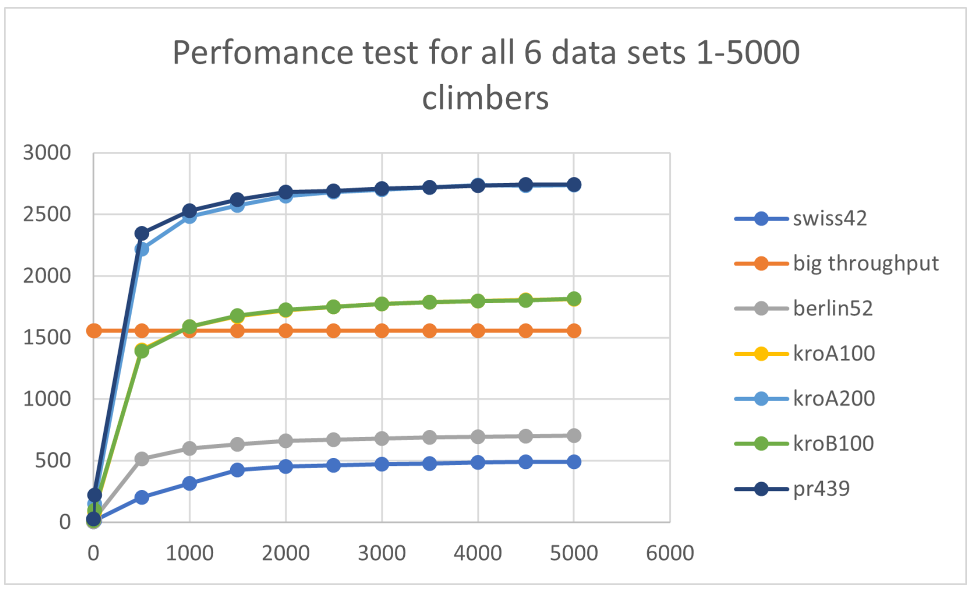


Figure 20: Throughput of all datasets for 1-5000 climbers

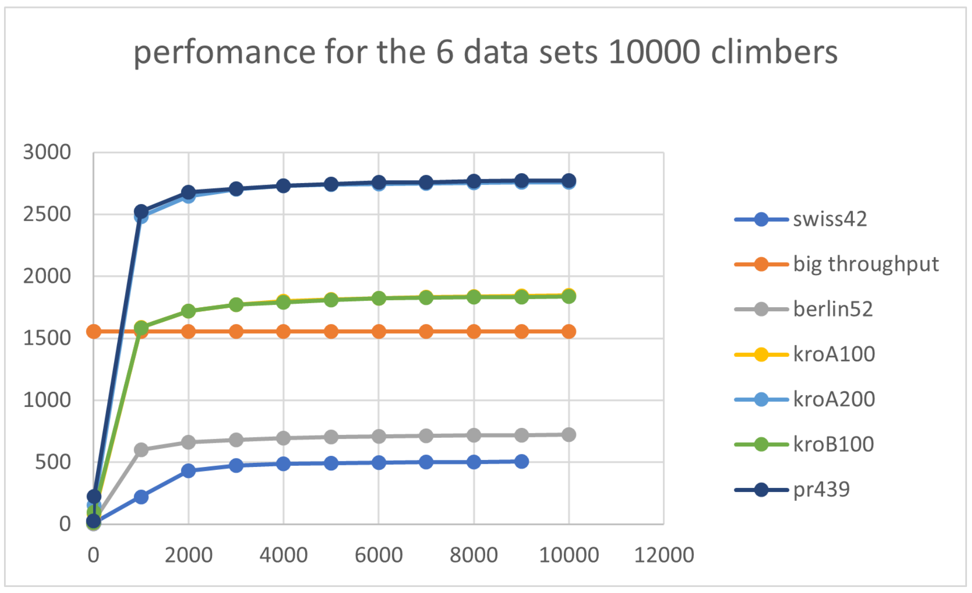


Figure 21: Throughput of all datasets for 1-10000 climbers

7.7.10 Discussuion of results

Throughout all the data sets it can be seen that the throughput increases when the number of climbers increases. Furthermore that the smaller data sets do not reach the big throughput of 1555 GB/s, but only get about 30% or 40% for the swiss42.tsp and berlin52.tsp data sets respectively. However the bigger data sets like pr439.tsp and kroA200.tsp gets a throughput that exceeds the big throughput. They both reach about 160% or 1.6 times the big throughput.

The reason the bigger data sets can exceed the big throughput is due to the vast majority of memory accesses are to the distance matrix that resides in global memory. Because the distance matrix for the biggest data set (pr439.tsp) consists of 439^2 entries, the uses $439^2 \cdot = 770.884$ bytes of space. This is only $\frac{770.884}{10^6} = 0.77MB$ of the space available in the L2 cache. This means that the distance matrix is able to reside in the 40 MB L2 cache, which is faster to access than global memory.
The reason that only the bigger data sets get a throughput that reaches (or exceeds) the big throughput, is that the number of while iterations for each climber, to find a local optimum is too small, for data sets with fewer cities. For the swiss42.tsp and berlin52.tsp data sets, each climber made 29 or 43 while loop iterations respectively. The kroA200 and pr439 did 150 and 236. The maximum throughput seems to be around 150 iterations of the while loop 22.

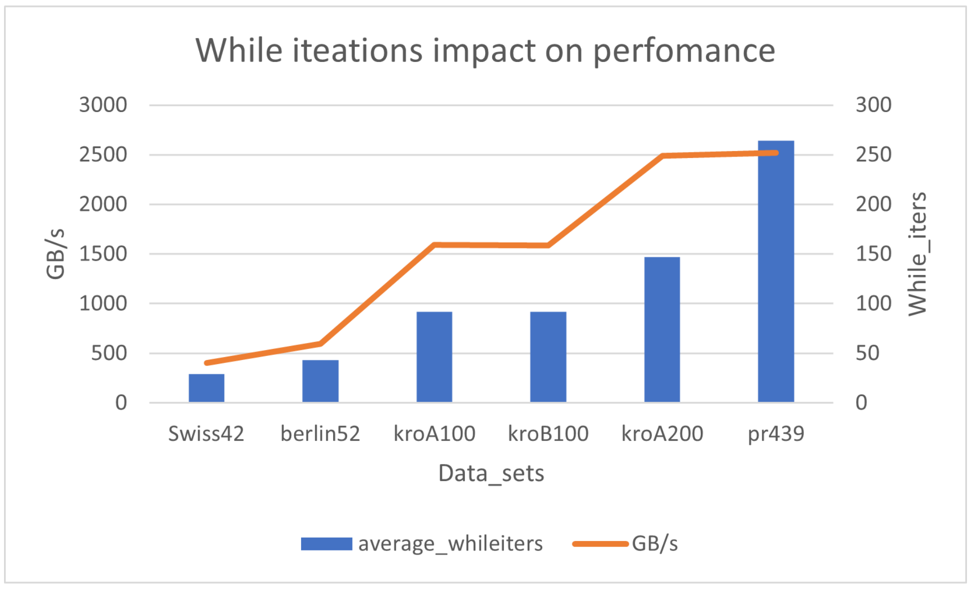


Figure 22: While iterations impact on performance

Table 4: Test results for 1000 climbers

	average_whileiters	**GB/s**
Swiss42	29	399.94
berlin52	43	596.84
kroA100	92	1592.35
kroB100	92	1584.71
kroA200	147	2487.95
pr439	264	2521.72

Another factor is that fewer accesses to global memory are made in each iteration for the smaller data sets, as this scales quadratically with the number of cities. Therefore the smaller data sets cannot exploit the temporal locality of the distance matrix to a high enough degree to reach the big throughput of the GPU.

8 Further Work

For future work on this project, the following would be interesting to look into:

8.1 Optimizing tour generation

The randomized tour kernel, see section 6.3.3, can be optimized by randomizing the tours on a local initial tour for each thread. Once the thread has randomized it on its local memory it will write it column-wise to the matrix (to achieve coalesced access to device memory). This way the threads will not randomize on the device memory, but on the local memory.

8.2 Handle more types of data sets

The CUDA implementation can only compute data sets of complete undirected graphs. It is therefore a limitation of the implementation that it can not compute data sets of directed graphs or incomplete graphs.
Furthermore, special cases of data like geographical data are not handled and support for these could be implemented.

8.3 Compare with other heuristic algorithms

There exist many more heuristic algorithms than the ones discussed in this project [**he**]. It would be interesting to analyze and compare algorithms such as:

- Genetic algorithms
- Particle swarm algorithm
- Tabu search
- artificial bee colony
- gradient based

8.4 Comparisons with other GPU versions

Our solutions should be compared to other GPU versions like GPU versions of ACO [3] and SA [31], to compare the quality of the solutions. This would also enable comparisons of data sets with more than 100 cities.

9 Conclusion

In this project, the TSP problem has been solved with a 2-Opt local search algorithm, implemented on GPU. A short survey of heuristic algorithms popular for solving TSP was done, and the 2-Opt local search algorithm was chosen, due to it not relying on parameters, and its potential for parallelization.
The algorithm was implemented in CUDA where each block was assigned one climber, such that both the outer parallelism of the grid and the inner parallelism of a block were exploited.
The implementation was compared with a similar implementation, that from O'Neil et al. [21], who assigned one thread to one climber, exploiting only the outer parallelism. The comparison showed that for a smaller number of climbers, our version had a significant speed-up of up to 27 times for 1000 climbers, but is slower for a greater number of climbers. However, it was evaluated that for most data sets no more than 1000 climbers were required to get a solution with a cost in the range of $> 95\%$ of the optimal cost, for most data sets.

To evaluate how well the solution used the hardware GPU, the throughput was measured. It was found that smaller data sets could not satisfy the GPU, with respect to the big throughput of the GPU, while data sets $>= 1000 cities$ exceeded the big throughput since that distance matrix was accessed more. The big throughput could be exceeded since the distance matrix fit in L2 memory, which is faster than global memory.

References

[1] NVIDIA Corporation & Affiliates. *CUDA C++ Programming Guide.* 2023. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#from-graphics-processing-to-general-purpose-parallel-computing-gpu-devotes-more-transistors-to-data-processing. (accessed: 06.06.2023).

[2] NVIDIA Corporation & Affiliates. *The GPU Devotes More Transistors to Data Processing.* 2023. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/_images/gpu-devotes-more-transistors-to-data-processing.png. (accessed: 06.06.2023).

[3] Hongtao Bai et al. "MAX-MIN Ant System on GPU with CUDA". In: *2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC).* 2009, pp. 801–804. DOI: 10.1109/ICICIC.2009.255.

[4] Bernd Bullnheimer, Richard Hartl, and Christine Strauss. "A New Rank Based Version of the Ant System - A Computational Study". In: *Central European Journal of Operations Research* 7 (Jan. 1999), pp. 25–38.

[5] Martin Burtscher and Molly A. O'Neil. *TSP_GPU11.cu.* 2011. URL: https://userweb.cs.txstate.edu/~burtscher/research/TSP_GPU/TSP_GPU11.cu. (accessed: 19.01.2022).

[6] G. A. Croes. "A Method for Solving Traveling-Salesman Problems". In: *Operations Research* 6 (1958), pp. 791–812.

[7] Daniel Delahaye, Supatcha Chaimatanan, and Marcel Mongeau. "Simulated annealing: From basics to applications". In: *Handbook of Metaheuristics.* Ed. by Michel Gendreau and Jean-Yves Potvin. Vol. 272. International Series in Operations Research & Management Science (ISOR). Springer, 2019, 8–9.ISBN 978-3-319-91085-7. DOI: 10.1007/978-3-319-91086-4_1. URL: https://hal-enac.archives-ouvertes.fr/hal-01887543.

[8] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. *Parallel Programming in Futhark.* 2018. URL: https://futhark-book.readthedocs.io.

[9] Yavuz Eren, İbrahim B. Küçükdemiral, and İlker Üstoğlu. "Chapter 2 - Introduction to Optimization". In: *Optimization in Renewable Energy Systems.* Ed. by Ozan Erdinç. Boston: Butterworth-Heinemann, 2017, pp. 49–53. ISBN: 978-0-08-101041-9. DOI: https://doi.org/10.1016/B978-0-08-101041-9.00002-8. URL: https://www.sciencedirect.com/science/article/pii/B9780081010419000028.

[10] Pradeep Gupta. *CUDA Refresher: The CUDA Programming Model.* 2020. URL: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/. (accessed: 08.06.2023).

[11] Pradeep Gupta. *Memory hierarchy in GPUs.* 2020. URL: https://developer-blogs.nvidia.com/wp-content/uploads/2020/06/memory-hierarchy-in-gpus-1.png. (accessed: 08.06.2023).

[12] Mark Harris. *How to Access Global Memory Efficiently in CUDA C/C++ Kernels.* 2013. URL: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/. (accessed: 08.06.2023).

[13] Mark Harris. *Optimizing Parallel Reduction in CUDA.* n.d. URL: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf. (accessed: 07.06.2023).

[14] Troels Henriksen et al. “Compiling Generalized Histograms for GPU”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’20. Atlanta, Georgia: IEEE Press, 2020. ISBN: 9781728199986.

[15] http://comopt.ifi.uni-heidelberg.de/. *TSPLIB95/tsp*. 1995. URL: `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/`. (accessed: 29.05.2023).

[16] David S. Johnson and Lyle A. McGeoch. “The Traveling Salesman Problem: A Case Study in Local Optimization”. In: 2008.

[17] Elemar Júnior. *Implementing Parallel Reduction in CUDA*. 2018. URL: `https://dev.to/elemarjr/implementing-parallel-reduction-in-cuda-3pbg`. (accessed: 08.06.2023).

[18] Selcuk Keskin, Omer Cetin, and Taşkın Koçak. “Real-Time FFT Computation Using GPGPU for OFDM-Based Systems”. In: *Circuits Systems and Signal Processing* 35 (June 2015). DOI: `10.1007/s00034-015-0106-5`.

[19] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. “Optimization by Simulated Annealing”. In: *Science* 220.4598 (1983), pp. 671–680. DOI: `10.1126/science.220.4598.671`. eprint: `https://www.science.org/doi/pdf/10.1126/science.220.4598.671`. URL: `https://www.science.org/doi/abs/10.1126/science.220.4598.671`.

[20] James McCaffrey. *Test Run - Ant Colony Optimization*. February 2012. URL: `https://learn.microsoft.com/en-us/archive/msdn-magazine/2012/february/test-run-ant-colony-optimization`. (accessed: 25.05.2023).

[21] Dan Tamir Molly A. O’Neiland and Martin Burtscher. *A Parallel GPU Version of the Traveling Salesman Problem*. 2011. URL: `https://userweb.cs.txstate.edu/~mb92/papers/pdpta11b.pdf`. (accessed: 19.01.2022).

[22] P. Munksgaard et al. “Memory Optimizations in an Array Language”. In: *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2022, pp. 424–438. URL: `https://doi.ieeecomputersociety.org/`.

[23] Yaghout Nourani and Bjarne Andresen. “A comparison of simulated annealing cooling strategies”. In: *Journal of Physics A: Mathematical and General* 31.41 (Oct. 1998), p. 8373. DOI: `10.1088/0305-4470/31/41/011`. URL: `https://dx.doi.org/10.1088/0305-4470/31/41/011`.

[24] Musa Peker, Baha Sen, and Pınar Kumru. “An efficient solving of the traveling salesman problem: The ant colony system having parameters optimized by the Taguchi method”. In: *TURKISH JOURNAL OF ELECTRICAL ENGINEERING & COMPUTER SCIENCES* 21 (Jan. 2013), pp. 2015–2036. DOI: `10.3906/elk-1109-44`.

[25] César Rego et al. “Traveling salesman problem heuristics: Leading methods, implementations and latest advances”. In: *European Journal of Operational Research* 211.3 (2011), pp. 427–441. ISSN: 0377-2217. DOI: `https://doi.org/10.1016/j.ejor.2010.09.010`. URL: `https://www.sciencedirect.com/science/article/pii/S0377221710006065`.

[26] Gerhard Reinelt. *TSPLIB 95*. 1995. URL: `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf`. (accessed: 13.12.2022).

[27] R. Schenck et al. “AD for an Array Language with Nested Parallelism”. In: *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2022, pp. 829–843. URL: `https://doi.ieeecomputersociety.org/`.

[28] TECHPOWERUP. *NVIDIA A100 PCIe 40 GB*. 2020. URL: `https://www.techpowerup.com/gpu-specs/a100-pcie-40-gb.c3623`. (accessed: 25.05.2023).

[29] Yong Wang and Zunpu Han. “Ant colony optimization for traveling salesman problem based on parameters optimization”. In: *Applied Soft Computing* 107 (2021), p. 107439. ISSN: 1568-4946. DOI: `https://doi.org/10.1016/j.asoc.2021.107439`. URL: `https://www.sciencedirect.com/science/article/pii/S1568494621003628`.

[30] Ingo Wegener. “Simulated Annealing Beats Metropolis in Combinatorial Optimization”. In: *Automata, Languages and Programming*. Ed. by Luís Caires et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 589–601.

[31] Kai-Cheng Wei, Chao-Chin Wu, and Hui-Liang Yu. “Mapping the Simulated Annealing Algorithm onto CUDA GPUs”. In: *2015 10th International Conference on Intelligent Systems and Knowledge Engineering (ISKE)*. 2015, pp. 358–365. DOI: `10.1109/ISKE.2015.97`.

[32] Xianmin Wei. “Parameters Analysis for Basic Ant Colony Optimization Algorithm in TSP”. In: *International Journal of u- and e- Service, Science and Technology* 7 (2014), pp. 159–170.

[33] Eric W Weisstein. *Hamiltonian Path*. URL: `https://mathworld.wolfram.com/HamiltonianPath.html`.

A Explanation of how to calculate the `i` and `j` values

In the original parallel version the `i` and `j` values are computed during initialization and written to device memory, see more in section 6.3.1. During the flatten nested loops the values are read from device memory, see more in section 6.3.4, subsection "Flattened nested `for` loops". This is expensive and therefore it would be desirable for each thread to compute the `i` and `j` values when they are needed in the flattened nested loops. This turned out to be possible and will be demonstrated by working with an example of 10 cities.

The following pseudo code is of the sequential nested `for` loops:

```
...   for(int i = 0; i < (cities - 2); i++) {
          for(int j = i+2; j < cities; j++){
              change = distM[tour[i]][tour[j]] + distM[tour[i+1]][tour[j+1]] -
              (distM[tour[i]][tour[i+1]] + distM[tour[j]][tour[j+1]]);
              if(change < minChange){
                  minChange = change; min_i = i; min_j = j;}}}...
```

By looking at the sequential pseudo code above and inserting $cities = 10$ the outer loop iterates $cities - 2 = 8$ times and the inner loop iterates $8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 36$ times. The total iterations are therefore 36 times and is the sum of $cities - 2$. For future calculations the total iterations, $totIter$, can be computed with the sum formula:

$$totIter = \frac{(cities - 2) * (cities - 1)}{2}$$

The values of `i` and `j` for the 36 iterations are listed below. Each subarray is a new iteration of the outer loop and therefore there are 8 subarrays.

$$\texttt{i} : [[0,0,0,0,0,0,0,0],\ [1,1,1,1,1,1,1],\ [2,2,2,2,2,2],\ [3,3,3,3,3],\ [4,4,4,4],\ [5,5,5],\ [6,6],\ [7]]$$

$$\texttt{j} : [[2,3,4,5,6,7,8,9],\ [3,4,5,6,7,8,9],\ [4,5,6,7,8,9],\ [5,6,7,8,9],\ [6,7,8,9],\ [7,8,9],\ [8,9],\ [9]]$$

id : 0 1 2 3 4 5 6 .... 21.........25..35

The id's that are listed above represents the thread ids in a block larger than 36 threads. This means that thread id 5 computes `i` $= 0$ and `j` $= 7$ and thread id 21 computes `i` $= 3$ and `j` $= 5$.
For the different threads to compute the correct `i` and `j` values each thread needs to first calculate which of the outer loop iterations it is part of, i.e. which subarray it is part of. In this case with $cities = 10$ thread id 0 to 7 are part of the first iteration of the outer loop and therefore the first subarray. Thread id 8 to 14 are part of the second iteration of the outer loop and second subarray. Thread id 15 to 20 are part of the third and so on.
Each subarray is increasing one in size compared to its subarray to the right. With this pattern in the subarrays it is possible to use the sum formula to compute which subarray a thread is part of.

$$\sum_{i=1}^{n} i = \frac{n \times (n+1)}{2}$$

In the example of finding the correct `i` and `j` values for thread id 21 it is needed to know how many iterations by the inner loop are still to be executed. This gives the formula:

$$restIter = totIter - id = 36 - 21 = 15$$

$restIter$ is used to calculate which subarray thread id 21 is part of by using the summation formula and finding which value n summed gives $restIter$.

$$\frac{n \times (n+1)}{2} = restIter = 15$$

$$n \times (n+1) = (15 \times 2)$$

$$n \times (n+1) - (15 \times 2) = 0$$

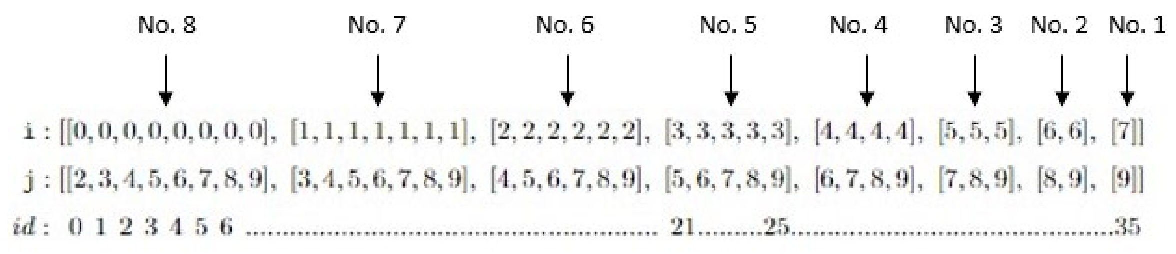
$$n^2 + n - (15 \times 2) = 0$$

The expression now has the form of a quadratic equation, $a \times x^2 + b \times x + c = 0$, and in the quadratic formula x is isolated with the formula $x = \frac{-b \pm \sqrt{b^2 - (4ac)}}{2a}$, therefore n can be isolated with this formula:

$$n = \frac{-1 \pm \sqrt{1 - (4 \times 1 \times (-(2 \times 15)))}}{2 \times 1}$$

$$n = (-6, 5)$$

The positive value 5 is the number of iterations that still needs to be executed by the outer loop and is also the 5th subarray (reading right to left) which thread id 21 is part of:



```
i : [[0,0,0,0,0,0,0,0], [1,1,1,1,1,1,1], [2,2,2,2,2,2], [3,3,3,3,3], [4,4,4,4], [5,5,5], [6,6], [7]]
j : [[2,3,4,5,6,7,8,9], [3,4,5,6,7,8,9], [4,5,6,7,8,9], [5,6,7,8,9], [6,7,8,9], [7,8,9], [8,9], [9]]
id :  0 1 2 3 4 5 6 ........................................................ 21.........25............................................35
```

Figure 23: Numbering the subarrays

Therefore it will only be the positive number of the quadratic formula that will be used.
The goal is now to get all the thread id's (id 21 to id 25) that are part of the 5th last iteration of the outer loop to compute the number 5 by using quadratic formula that is used above. When thread id 25 uses the quadratic formula the result is:

$$n_{pos} = \frac{-1 + \sqrt{1 - (4 \times 1 \times (-(2 \times (totIter - id))))}}{2 \times 1}$$

$$n_{pos} = \frac{-1 + \sqrt{1 - (4 \times 1 \times (-(2 \times (36 - 25))))}}{2 \times 1}$$

$$n_{pos} = 4.2170$$

Since this is less than 5 there will be added 0.9999 to n_{pos} and then the floor function will be applied in order to get the result 5 in this case. The formula to compute which iteration of the outer loop a thread id is part of n_{curr} is therefore:

$$n_{curr} = \lfloor \frac{-1 + \sqrt{1 - (4 \times 1 \times (-(2 \times (totIter - id))))}}{2 \times 1} + 0.9999 \rfloor$$

To verify the formula n_{curr} the formula is computed with thread id 21 and thread id 25, which should result in 5, and thread id 6 which should result in 8. See figure 23 to verify.
Thread id 21:

$$n_{curr} = \lfloor \frac{-1 + \sqrt{1 - (4 \times 1 \times (-(2 \times (36 - 21))))}}{2 \times 1} + 0.9999 \rfloor = 5$$

Thread id 25:

$$n_{curr} = \lfloor \frac{-1 + \sqrt{1 - (4 \times 1 \times (-(2 \times (36 - 25))))}}{2 \times 1} + 0.9999 \rfloor = 5$$

Thread id 6:

$$n_{curr} = \lfloor \frac{-1 + \sqrt{1 - (4 \times 1 \times (-(2 \times (36 - 6))))}}{2 \times 1} + 0.9999 \rfloor = 8$$

The calculations are correct.

Now that each thread can calculate which subarray is it part of the value `i` can be determined. As previously stated there are $cities - 2 = 8$ iterations in total of the outer loop, for $cities = 10$. By subtracting this number with n_{curr}, the value `i` is computed.

$$\texttt{i} = (cities - 2) - n_{curr}$$

To verify, the value `i` is computed by thread id 21, thread id 25 and thread id 6.
Thread id 21:

$$\texttt{i} = (cities - 2) - n_{curr} = (10 - 2) - 5 = 3$$

Thread id 25:

$$\texttt{i} = (cities - 2) - n_{curr} = (10 - 2) - 5 = 3$$

Thread id 6:

$$\texttt{i} = (cities - 2) - n_{curr} = (10 - 2) - 8 = 0$$

When comparing the results with figure 23, where it is shown which `i` value belongs to which thread id, it is possible to verify that the results `i` are correct.

Now that each thread can calculate the correct `i` value, it also has to calculate the correct `j` value. The `j` value is computed based on the `i` value. When looking above at the sequential pseudo code for the nested `for` loops the first iteration of the inner loop will be $\texttt{j} = \texttt{i} + 2$. This means that the first thread of each subarray in figure 23 can easily compute $\texttt{j} = \texttt{i} + 2$. As an example in the figure thread id 21, which is the first thread in subarray no. 5, can compute $\texttt{j} = \texttt{i} + 2 = 3 + 2 = 5$. However the second thread in a subarray will compute $\texttt{j} = \texttt{i} + 2 + 1$, the third $\texttt{j} = \texttt{i} + 2 + 2$ and so on. Thread id 25 in figure 23 will have to compute $\texttt{j} = \texttt{i} + 2 + 4 = 3 + 2 + 4 = 9$.

In order to compute the `j` value the first thread id of each subarray must be computed. By subtracting the first thread id in a subarray from the current thread id it is possible to compute the number of iterations by the inner loop, *iter*, there are between the first thread and the current thread. When the number *iter* is found it can be added to the formula to find `j`:

$$\texttt{j} = \texttt{i} + 2 + iter$$

The first thread id, fti, is computed based on the current subarray that the thread is part of, n_{curr}, with the summation formula:

$$fti = totIter - \frac{n_{curr} \times (n_{curr} + 1)}{2}$$

As an example, for thread id 25 it is expected that $fti = 21$.

$$fti = totIter - \frac{n_{curr} \times (n_{curr} + 1)}{2} = 36 - \frac{5 \times (5 + 1)}{2} = 21$$

This is correct. Now that it is possible to calculate the first thread id in each subarray, the value *iter* can be computed by subtracting the current thread id with the first thread id.

$$iter = threadId - fti$$

By inserting the information for thread id 25 *iter* will be:

$$iter = 25 - 21 = 4$$

Now the value `j` can be computed for thread 25:

$$\texttt{j} = \texttt{i} + 2 + iter = 3 + 2 + 5 = 9$$

When comparing to figure 23 it correct.

Following is a summary of the formulas created:
The formula to compute $totIer$:

$$totIter = \frac{(cities - 2) * (cities - 1)}{2}$$

The formula to compute the value `i`:

$$n_{curr} = \lfloor \frac{-1 + \sqrt{1 - (4 \times 1 \times (-(2 \times (totIter - threadId))))}}{2 \times 1} + 0.9999 \rfloor$$

$$\texttt{i} = (cities - 2) - n_{curr}$$

The formula to compute the value `j`:

$$\texttt{j} = \texttt{i} + 2 + (threadId - (totIter - \frac{n_{curr} \times (n_{curr} + 1)}{2}))$$

These formulas are inserted into the program of the calculated version and can be seen in the following codesnippet, which is from the `tsp-kernels.cu.h` file, function `twoOptKerCalculated()`. Line 4 computes the discriminant of the quadratic formula used to compute the value `i`. Line 5 computes the quadratic formula and adds 0.9999. Line 6 takes the floor and now n_{curr} is computed. Line 7 computes the value `i`. Line 8 computes the value `j`.

```
// 2 opt move - calculated version
float tmp;
for(int ind = idx; ind < totIter; ind += block_size){
    d = 1-(4*(-2*(totIter-ind)));
    tmp = ((-1+(sqrt((float) d)))/2)+0.9999;
    curr = (int) tmp;
    i = (cities-2) - curr;
    j = (i+2) + (ind-(totIter-((curr*(curr+1))/2)));
    ip1 = i+1;
    jp1 = j+1;
    change = glo_dist[tour[i]*cities+tour[j]] +
            glo_dist[tour[ip1]*cities+tour[jp1]] -
            (glo_dist[tour[i]*cities+tour[ip1]] +
            glo_dist[tour[j]*cities+tour[jp1]]);

    //Each thread shall hold the best local change found
    ChangeTuple check = ChangeTuple(change,(unsigned short)i, (unsigned short) j);
    localMinChange = minInd::apply(localMinChange,check);
}
```

B Code for counting memory accesses

Copying from global memory

```
.
.
.
    //copy global tour to shared memory
    for(int t = idx; t < cities+1; t += block_size){
        tour[t] = glo_tours[blockIdx.x * (cities+1) + t];
        // cities + 1 * 2 bytes
    }
```

Finding best change using global memory

```
.
.
.
    while(minChange[0].change < 0){
.
.
        for(int ind = idx; ind < totIter; ind += block_size){
        .
        .
        .
            change = glo_dist[tour[i]*cities+tour[j]] +
                    glo_dist[tour[ip1]*cities+tour[jp1]] -
```

```
                        (glo_dist[tour[i]*cities+tour[ip1]] +
                        glo_dist[tour[j]*cities+tour[jp1]]);
               // 4 * 4 bytes
.
.
.
```

calculating cost of tour

```
.
.
.
    int idx = threadIdx.x;
    int block_size = blockDim.x;
    int sum = 0;
    int glo_i, glo_ip1;
    for(int i = idx; i < cities; i += block_size){
        glo_i = lo_tour[i];
        glo_ip1 = lo_tour[i+1];
        sum += glo_dist[glo_i * cities + glo_ip1];
    }
.
.
.
```

copying back to global memory

```
        //copy best local tour from shared memory to global memory
    for(int t = idx; t < cities+1; t += block_size){
        glo_tours[blockIdx.x * (cities+1) + t] = tour[t];
    }
    // 4 * cities+1 bytes

    //Writing local optimum results to global memory
    if(idx == 0){
        glo_result[blockIdx.x * 2] = local_opt_cost;
        glo_result[blockIdx.x * 2+1] = blockIdx.x;
        counter[blockIdx.x] = while_block[0];
        //counter[blockIdx.x * 2 + 1]  = blockIdx.x;
        //printf("number of while iters in block %d is : %d  \n", blockIdx.x, while_blo
    }
    // 4 * 2
```