CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Science

Bachelor's Thesis

GPU Parallelization of the Backtracking Algorithm for Single-vehicle DARP

Lukáš Kulhánek
Open Informatics

May 2021
Supervisor: Ing. David Fiedler

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kulhánek** Jméno: **Lukáš** Osobní číslo: **474407**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Software**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Paralelizace backtracking algoritmu pro Single-vehicle DARP s pomocí grafické karty

Název bakalářské práce anglicky:

GPU Parallelization of the Backtracking Algorithm for Single-vehicle DARP

Pokyny pro vypracování:

Seznam doporučené literatury:

[1] T. Carneiro Pessoa, J. Gmys, F. H. de C. Júnior, N. Melab, and D. Tuyttens, 'GPU-accelerated backtracking using CUDA Dynamic Parallelism', Concurrency and Computation: Practice and Experience, vol. 30, no. 9, p. e4374, 2018, doi: 10.1002/cpe.4374.
[2] R. Finkel and U. Manber, 'DIB - a distributed implementation of backtracking', ACM Trans. Program. Lang. Syst., vol. 9, no. 2, pp. 235–256, Mar. 1987, doi: 10.1145/22719.24067.
[3] V. N. Rao and V. Kumar, 'On the efficiency of parallel backtracking', IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 4, pp. 427–437, Apr. 1993, doi: 10.1109/71.219757.
[4] E. Speckenmeyer, B. Monien, and O. Vornberger, 'Superlinear speedup for parallel backtracking', in Supercomputing, Berlin, Heidelberg, 1988, pp. 985–993, doi: 10.1007/3-540-18991-2_58.
[5] J. Jenkins, I. Arkatkar, J. D. Owens, A. Choudhary, and N. F. Samatova, "Lessons Learned from Exploring the Backtracking Paradigm on the GPU," in Euro-Par 2011 Parallel Processing, Berlin, Heidelberg, 2011, pp. 425–437, doi: 10.1007/978-3-642-23397-5_42.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. David Fiedler, centrum umělé inteligence FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **04.03.2021** Termín odevzdání bakalářské práce: _____

Platnost zadání bakalářské práce: **19.02.2023**

Ing. David Fiedler podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

Acknowledgement / Declaration

I would like to thank to my thesis supervisor Ing. David Fiedler for his guidance and helpful input.

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 21. 05. 2021

.....

Abstrakt / Abstract

Tato práce se zabývá paralelizací backtracking algoritmu pro řešení jednovozidlového Dial-a-ride problému. Jednovozidlový Dial-a-ride problém je dobře známý teoretický problém, ve kterém je vozidlo použito k obsluze zákazníků. Každý zákazník má přiřazeny dva body (místo vyzvednutí a místo vyložení) a každý z těchto bodů má přiřazeno časové okno, ve kterém musí vozdilo tento bod navštívit. Tento algoritmus je používán v projektu darp-benchmark, kde je používán v rámci Vehicle Group Assignment (VGA) algoritmu. Jsou představena dvě řešení - paralelní algoritmus pro běh na CPU a paralelní algoritmus pro běh na GPU. Zlepšení v rychlosti paralelních verzí je změřeno a porovnáno oproti původní sériové verzi.

Klíčová slova: dial-a-ride, paralelizace, CUDA, OpenMP

Překlad titulu: Paralelizace backtracking algoritmu pro Single-vehicle DARP s pomocí grafické karty

This thesis aims to design and implement a parallel version of a backtracking algorithm used for solving Single vehicle Dial-a-ride problem. The Single vehicle Dial-a-ride problem (DARP) is a well-known problem in which a vehicle is used to serve customers' requests. Each user transportation request has two points associated with it (pickup and delivery point) and each point has a specific time window within which the point must be visited by a vehicle. The backtracking SVDARP solver is used in the darp-benchmark project as a part of the Vehicle Group Assignment (VGA) algorithm. The VGA algorithm solves the general Dial-a-ride by decomposing it into many SVDARPs. A GPU parallel and a CPU parallel approaches to the solution are proposed and implemented. The performance of the new algorithms is measured and compared to the original SVDARP solver.

Keywords: dial-a-ride; parallelization; CUDA; OpenMP

/ Contents

Tables / Figures

Chapter 1
Introduction

On-demand transportation has become quite popular in today's world. It offers an alternative to fixed schedule/route transportation services. Examples of such on-demand models are carsharing, bikesharing or ridesharing.

Ridesharing is a service that provides a one-way transportation on a short notice. Instead of having vehicles go through regular routes at specific times, the customers send requests specifying where and when they want to be picked up and dropped off. Some of the more notable ridesharing companies are Uber and Lyft. Possible benefits compared to the standard means of transportation are cost and time optimizations, lesser environmental impact or increased passenger comfort. As the routes of the vehicles are not known in advance a way to determine the best possible order in which to serve the customers is needed. This problem is known as the dial-a-ride problem.

The general dial-a-ride problem (DARP) is a well-known problem in which a fleet of vehicles is used to serve customers' requests. Each user transportation request has two points associated with it (pickup and delivery point) and each point has a specific time window within which the point must be visited by a vehicle. Limitations on how much time the user spends in the vehicle and how long the vehicle can travel are usually also imposed. The goal is to construct routes for the vehicles in such a way that all aforementioned constraints are satisfied while keeping the distance travelled by the vehicles minimal.

Since the DARP was formulated, many different algorithms used for solving it were proposed. The DARP is considered to be NP-hard [1], therefore an exact solution usually cannot be found in a practical time and the solution is usually approximated using heuristic algorithms [2] [3] [4]. However, an exact algorithm called Vehicle Group Assignment (VGA) [5] was recently proposed. The algorithm can be used for solving DARP. It decomposes the main problem into solving many single-vehicle dial-a-ride problems (SVDARP) for which the exact solution can be established in a reasonable time.

The VGA algorithm is implemented in the darp-benchmark[1] project. It uses a backtracking algorithm for solving the SVDARP subproblems. The time complexity of the most commonly used DARP evaluation scheme for determining the feasibility and the cost of a solution proposed by [6] is known to be $\mathcal{O}(n^2)$ where n is the number of nodes in the plan [1]. Due to that complexity the time spent solving SVDARP instances makes up a significant part of the execution time of the VGA implementation in the darp-benchmark project. Achieving a significant speedup of the algorithm solving the SVDARP would therefore improve the overall time performance of the algorithms using it dramatically.

This thesis proposes two approaches to speeding up the backtracking algorithm. The first one uses the CUDA framework to solve the SVDARP in parallel using GPU. The second approach uses the OpenMP library to achieve parallelization using CPU threads. The approaches are described in Chapter 4 and their implementation in the project is

[1] https://gitlab.fel.cvut.cz/fiedlda1/darp-benchmark

described in Chapter 5. The results and parameter tuning are presented in Chapter 6. The remainder of this thesis is organized as follows. In Chapter 2 the SVDARP is formally defined. The Chapter 3 presents the work related to solving SVDARP and solving backtracking problems in parallel.

Chapter 2
Problem Specification

The dial-a-ride problem is an extension of the Pickup and Delivery Problem With Time Windows (PDPTW)[7] which falls under the class of Vehicle Routing Problem (VRP) [8]. The SVDARP is a special case of DARP where only one vehicle is available. It is defined on a complete weighted digraph $G = (V, E)$ and a set of n transportation requests. The set $V = \{0, 1, ..., 2n, 2n+1\}$ is the set of nodes where nodes 0 and $2n+1$ represent the depot, For a transportation request i it's pickup point is represented by node i and it's delivery point by node $n+i$. For each node $j \in V$, the interval $[e_j; l_j]$ is the associated time window: e_j is the earliest possible starting time and l_j is the latest possible starting time, d_j is the service time and q_j is the number of persons to transport. Given an edge $(i, j) \in E$, t_{ij} is the transportation time and c_{ij} is the transportation cost. The vehicle can hold up to Q persons at a time [1].

Given a node $i \in V$, following variables are specified:

- A_i the arrival time of the vehicle at the node
- B_i beginning of service time at the node
- $D_i = B_i + d_i$ departure time from the node
- $W_i = B_i - A_i$ waiting time at the node

For a transportation request i, $R_i = B_{n+i} - D_i$ is the total ride time. A solution s can be defined as a permutation node list λ such that every node $i \in \{1, 2...n\}$ is used. $\lambda(i)$ then denotes the i-th node visited on the vehicle trip. The first and last nodes in the trip must be the depot nodes so that $\lambda(0) = 0$, $\lambda(n+1) = n+1$. The total cost c of a solution s is given as $c(s) = \sum_{i=0}^{n} c_{\lambda(i)\lambda(i+1)}$ where $c_{\lambda(i)\lambda(j)}$ is the cost of travelling from node $\lambda(i)$ to node $\lambda(j)$. The objective of the SVDARP is to find the minimal total cost. A valid solution must satisfy the following constraints:

- the vehicle must carry at most Q persons at any point of the trip
- the beginning of service $B_i \in [e_i; l_i]$
- the ride time R_i does not exceed maximum allowed ride time L for all requests i
- the trip duration for the vehicle does not exceed the maximum allowed time

Chapter 3
State Of The Art

In this chapter a definition of backtracking is established. An overview of the published methods for parallelizing backtrack search is provided aswell as a short summary of some proposed methods for solving SVDARP.

Backtracking is a general algorithm used for solving various computational problems, most notably constraint satisfaction problems [9]. It is a complete algorithm meaning that a solution is guaranteed to be found if any exists. Problems that are solved using backtrack search generally have a property (such as cost) that can be used to decide which nodes in the search tree are worth to be explored and which should be abandoned. If a node is abandoned the algorithm backtracks to a point where a new search can be started [10]. The nodes in the search tree are usually explored in a depth-first fashion as finding a solution early can affect how early are the non-optimal nodes abandoned. A typical example of a problem solved using the backtrack search is the N-Queens problem. In the N-Queens problem a chessboard of NxN squares is considered and N queens need to be placed on the chessboard in such a way that no two queens can attack each other [11].

The backtrack search is primarily used for solving NP-hard computatinal problems which have exponential time complexity. Many approaches to improving the performance were proposed ranging from algorithmic improvements to the sequential algorithm, parallelization using distributed systems and, in the recent years, parallelization using GPU frameworks.

3.1 Parallel Indexed Search Tree Algorithm

The algorithm presented in [12] aims to improve scalability of backtracking algorithms solving any general NP-hard graph problem. The search space is represented using indexed search trees. Each node in the search tree has a weight assigned representing an estimated computation time required. The tasks are then distributed between the processing nodes according to the weights. Linear speed gains were observed even in systems with tens of thousands of cores.

3.2 GPU Parallel Tree Searching

The approach presented in [13] is an idea based on the limitations of the available GPU frameworks. The jobs cannot be distributed on the fly and additionally, the communication cost is high. The search tree is divided into two parts based on depth. The nodes starting from the root up to a depth s are processed sequentially and all the valid nodes at depth s are stored into an array/list. After that the nodes are copied over to the GPU and processed in parallel. If the number of nodes at depth s exceeds the number of threads available on the GPU using a queue on the GPU is proposed. The GPU loads the tasks (represented by nodes found in the sequential search) from the queue until all are completed.

3.3 GPU-based Integer-Vector-Matrix (IVM) Algorithm

Integer-Vector-Matrix is a data structure used for representing the search tree of permutation problems. The integer I in the data structure represents the depth of the next subproblem. The vector V points to the nodes present in the current permutation. The matrix M contains the nodes that remain to be visited at each depth. The IVM data structure is well suited for use in GPU processing as the memory requirements are known in advance, minimizing the need for dynamic memory allocation. Work stealing is then used for load balancing. When a thread completes its work, it increases a counter in the global memory by one. If a certain threshold is reached, all threads will pause exploring the search space and the work-stealing phase starts. Threads with completed IVMs will steal a portion of the interval to be explored in threads with incomplete IVMs. Exploration of the search space then continues again until all IVMs are completed [14].

3.4 Buffered Workpool

The Buffered Work-Pool approach is a hybrid dynamic load balancing technique that combines threading and message passing between the different processors of any cluster. The technique is based on the Master-Worker paradigm. The master prepares the initial tasks specific to the problem and distributes them to the workers. After that the master's goal is to communicate with the workers. The workers can request the master to either get more tasks or share some of its tasks with other workers[15].

3.5 GPU Dynamic Parallelism

The algorithms [16] [17] and [18] use a recently introduced feature of the NVIDIA CUDA framework called Dynamic Parallelism. This allows new kernels (functions run on GPU) to be launched from kernels that were already running without any need of CPU intervention so that overhead is minimized. A finer control of load-balancing can be achieved by dynamically launching new kernels on-demand with a possibility to combine multiple search strategies.

3.6 Methods for Solving SVDARP

In [19] the SVDARP was formulated as a integer linear program and a forward dynamic programming algorithm was proposed. At the first iteration the states represent a route from the depot to a first pickup node. At each subsequent iteration the new states are constructed from the previous states by adding another pickup or dropoff node to the route. At the last iteration the route has to end at the depot node again. The efficiency is further improved upon by eliminating states which violate the capacity, precedence or time constraints. The proposed algorithm has $\mathcal{O}(n^2 \cdot 3^n)$ complexity.

A minimum spanning tree algorithm is proposed in [20]. First it starts by constructing a Travelling Salesman (TS) tour through all the user nodes based on their minimal spanning tree. Then it chooses any node that is a pickup node and moves clockwise until all points are visited. At this point a local interchange operator [21] [22] can be performed on the TS tour. These steps (except for the first one) are then repeated for every pickup node and the tour with minimum length is chosen.

An adaptive insertion algorithm for solving the SVDARP is proposed in [23]. It proposes a new insertion heuristic, which operates as follows. For each customer it performs an insertion of the new customer to all feasible service sequences with respect to existing customers. The heuristic then determines the set of feasible service sequences with respect to the new customer and existing customers. This approach provides a globally optimal solution and has reasonable execution time for smaller instances. For larger instances an extension to the heuristic is proposed. For each user only a specified number of service sequences is explored. This reduces the execution time but also removes the guarantee of finding a globally optimal solution.

Chapter 4
Proposed Solution

When choosing an approach to the parallelization, various factors were taken into consideration. Although parallelization using one of the distributed computing algorithms could provide superior performance for very large instances, it also requires the algorithm to be run on a cluster with many computing nodes which is fairly impractial from a convenience standpoint. Using a GPU seems to be a better choice as many computers come equipped with GPUs already. The speedups presented in [16] [17] and [18] seem to be significant enough to provide the necessary speed boost. Therefore a multistage algorithm based on work presented in [16], [17] and [24] is proposed in this section. A CPU parallel version is also proposed using the OpenMP tasks [25].

4.1 Serial Algorithm

The current implementation uses a backtracking algorithm to solve the SVDARP. It starts with an empty plan and builds the solution up in a depth-first fashion. A first unused user request node is placed in the current position in the plan. The plan is then checked for feasibility. If the plan is feasible, the algorithm moves to the next position in the plan and attempts to place nodes at that position. If the plan is not feasible, the algorithm removes the user request node it placed at the position and tries placing another user request node instead. When all the user request nodes were placed at the current position or the plan was completed, the algorithm backtracks, removing the node at the current position and moving one position back in the plan. A complete plan with the minimal cost is returned as the final best solution. The algorithm is also presented in the Implementation chapter as Algorithm 5.1.

4.2 CPU Parallel Algorithm

The CPU parallel version of the algorithm follows the same principles as described in Section 4.1. The parallelization occurs every time a new incomplete plan is found. If the plan is feasible a new parallel task is created that explores the next position in the plan. This occurs up to a specified position in the plan at which no new task is created and the remainder of the plan is explored sequentially in the current task instead. When a new best plan is found replacing the old one should be done in a critical section to prevent a race condition as done in Algorithm 5.3.

4.3 GPU Parallel Algorithm

A visual summary of the main components of the algorithm can be seen in Figure 5.2. The first step of the algorithm is to explore the search space up to depth `max_depth` using the same approach as described in Section 4.1. All feasible partial solutions found

at this depth are stored for later processing as described in Algorithm 5.4. This is done using a modified version of the serial algorithm described in the Section 4.1.

The required amount of memory on the GPU is allocated beforehand as doing so during the kernel execution presents a significant performance hit. Therefore the host code has to determine the required memory size and allocate it on the GPU accordingly. The following memory requirements are made by the algorithm:

- An array where the feasible nodes discovered by the initial serial search are stored
- An array where each thread stores the best found result and the cost
- An array where each block stores the best found solution and the cost
- Space to fit the data associated with the SVDARP instance
- Space to fit the stack of each thread

An important aspect to take into consideration is the total memory available on the GPU. If the memory requirements are greater than the GPU can handle at once, the problem can be split and solved in multiple runs as described in [16].

Once the memory requirements have been established and the memory has been alocated (as in Algorithm 5.5) the algorithm then proceeds to the next phase. The partial solutions are copied to the GPU and a GPU kernel is launched as in Algorithm 5.6. The kernel (Algorithm 5.7) should be configured to run one thread per partial solution. The thread block size of the kernel can be set to a value of up to 1024. Each thread stores its best solution separately. Once all threads have finished running the master thread of the thread block then iterates through all the solutions found by the threads in the block and saves the best one as the block's best solution. After that the final result is retrieved by the host code. The block solutions are examined and the solution with the best cost is chosen as the final solution of all blocks.

Chapter 5
Implementation

The implementation of the proposed solution was integrated into the darp-benchmark[1] project. The CPU parallel solution was implemented using the OpenMP library and C++. The GPU parallel solution was implemented using the CUDA framework and C++. For clarity the algorithms are presented in a simplified form and many language specifics and optimizations are ommited.

5.1 Data Representation

Various classes used for representing the data were already present in the darp-benchmark. Their simplified versions are used in the descriptions of the algorithms in this chapter.

`Class VehiclePlan`

- `int arrival_time` - the time at which the vehicle arrives at the depot
- `int departure_time` - the time at which the vehicle leaves the depot
- `int cost` - cost of the plan
- `std::vector<PlanNode> nodes` - vector of the user nodes present in the plan

`Class PlanNode`

- `int arrival_time` - the time at which the vehicle arrives at the user node
- `int departure_time` - the time at which the vehicle departs from the user node
- `int node_id` - the index of the associated user node

`Class InstanceData`

- `int instance_size` - total count of the user nodes

5.2 Serial Algorithm

The implementation of the serial algorithm was already present in the darp-benchmark project. The algorithm uses a general recursive backtracking approach to solve the problem. It iterates through all the user requests present in the SVDARP instance. Then it inserts each user request currently not present in the vehicle plan at the given position and marks it as used. If the solution is feasible the algorithm recursively calls itself for position + 1 if the vehicle plan is not complete. When the vehicle plan has been completed (a leaf in the search tree was reached) it compares the final cost of the plan to the best found solution and saves it as the best found solution if the cost is better. If the generated solution is not feasible the user request is declared unused and the algorithm proceeds to the next user request.

[1] https://gitlab.fel.cvut.cz/fiedlda1/darp-benchmark

```
void svdarp_serial(VehiclePlan& plan, std::vector<int>& used_nodes,
                   InstanceData& data, VehiclePlan& best_plan,
                   int position = 0)
{
    for (int i = 0; i < data.request_nodes.size(); ++i)
    {
        if (used_nodes[i] == -1)
        {
            plan.nodes[position].node_id = i;
            used_nodes[i] = position;
            bool is_feasible = evaluate(plan, used_nodes, data);

            if (is_feasible)
            {
                if (position == data.instance_size + 1)
                {
                    if (plan.cost < best_plan.cost)
                        best_plan = plan;
                }
                else
                    svdarp_serial(plan, used_nodes, data, position + 1);
            }
            used_nodes[i] = -1;
        }
    }
}
```

Algorithm 5.1. Serial SVDARP

The `svdarp_serial` function is initially launched with an empty plan, vector of integers of size `data.instance_size` named `used_nodes`, data associated with the instance being solved and another empty plan `best_plan` with its cost set to the maximal possible value in the C++ language, as parameters. When the function finishes, `best_plan` will contain either the vehicle plan with the minimal cost or an empty plan in case no feasible solution was found.

5.3 CPU Parallel Algorithm

The CPU parallel version is realized using the OpenMP library. The parallelization is done using the OpenMP task paradigm. The algorithm is launched using the `svdarp_parallel_launch` function which accepts the SVDARP instance data as an argument. A variable representing the best solution, an empty plan and a vector of integers is defined initially the same way as described in the Serial Algorithm. A parallel OpenMP section then starts in which the `svdarp_parallel_search` function is launched on a single thread.

```
VehiclePlan svdarp_parallel_launch(InstanceData& instance)
{
    VehiclePlan plan;
    VehiclePlan best_plan;
    best_plan.cost = NUMERIC_MAX;
    std::vector<int> used_nodes;

    #pragma omp parallel
```

```
    {
        #pragma omp single
        svdarp_parallel_search(instance, plan, used_nodes, best_plan);
    }
    return best_plan;
}
```

Algorithm 5.2. Parallel CPU SVDARP launcher

The `svdarp_parallel_search` is a modified version of the Serial Algorithm. To prevent a race condition the comparison of the costs and assignment of the best plan (if a better one is found) is done in an OpenMP critical section. The consequent calls to the function are launched as a OpenMP tasks.

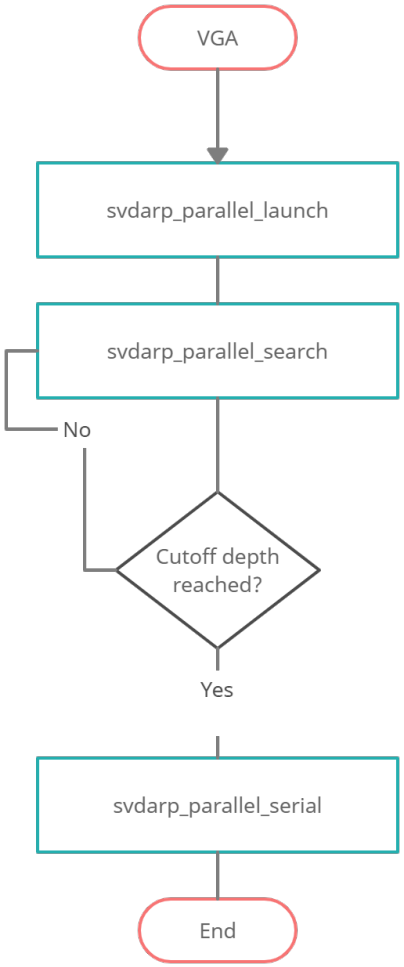

Figure 5.1. Visualization of the CPU parallel algorithm

```
void svdarp_parallel_search(InstanceData& instance, VehiclePlan plan,
                            std::vector<int> used_nodes,
                            VehiclePlan& best_plan, int position = 0)
{
    for (int i = 0; i < data.request_nodes.size(); ++i)
    {
```

```
            if (used_nodes[i] == -1)
            {
                plan.nodes[position].node_id = i;
                used_nodes[i] = position;
                bool is_feasible = evaluate(plan, used_nodes, data);

                if (is_feasible)
                {
                    if (position == data.instance_size + 1)
                    {
                        #pragma omp critical
                        {
                            if (plan.cost < best_plan.cost)
                                best_plan = plan;
                        };
                    }
                    else {
                        if (position < PARALLEL_DEPTH_END) {
                            #pragma omp task priority(position + 1)
                            svdarp_parallel_search(instance, plan,
                                                   used_nodes, best_plan,
                                                   position + 1);
                        }
                        else
                            svdarp_parallel_serial(instance, plan,
                                                   used_nodes, best_plan,
                                                   position + 1);
                    }
                }
                used_nodes[i] = -1;
            }
        }
    }
```

Algorithm 5.3. Parallel SVDARP CPU search

The `svdarp_parallel_serial` function is identical to the `svdarp_parallel_search` function except the arguments `plan` and `used_nodes` are now passed as reference for better performance. Any call of the function `svdarp_parallel_serial` is never launched as a task.

A deep copy of the variables `plan` and `used_nodes` is performed at every consequent call. In C++ a deep copy is done implicitly as the variables are passed to the function as values and not as references as in the serial version. At a specified depth the function is no longer launched as a task to mitigate the impact of the overhead of launching OpenMP tasks.

5.4 GPU Parallel Algorithm

The implementation follows the approach outlined in Chapter 4. The limitations of the capabilities of the CUDA framework have influenced the way the algorithm was implemented. The first issue was that although the CUDA framework supports the

C++ the C++ Standard Library (STL) is not present at all. Although an alternative library called Thrust exists and enables the basic functionality of the STL to be used in the GPU code, its performance has been described as lacking in the official CUDA forums. Another issue that complicated the implementation a lot is that CUDA is unable to perform deep copy of data to the GPU. Therefore special data structures that use only basic data types available in the C++ and a way to convert them to the data structures used in the rest of the darp-benchmark project had to be implemented.

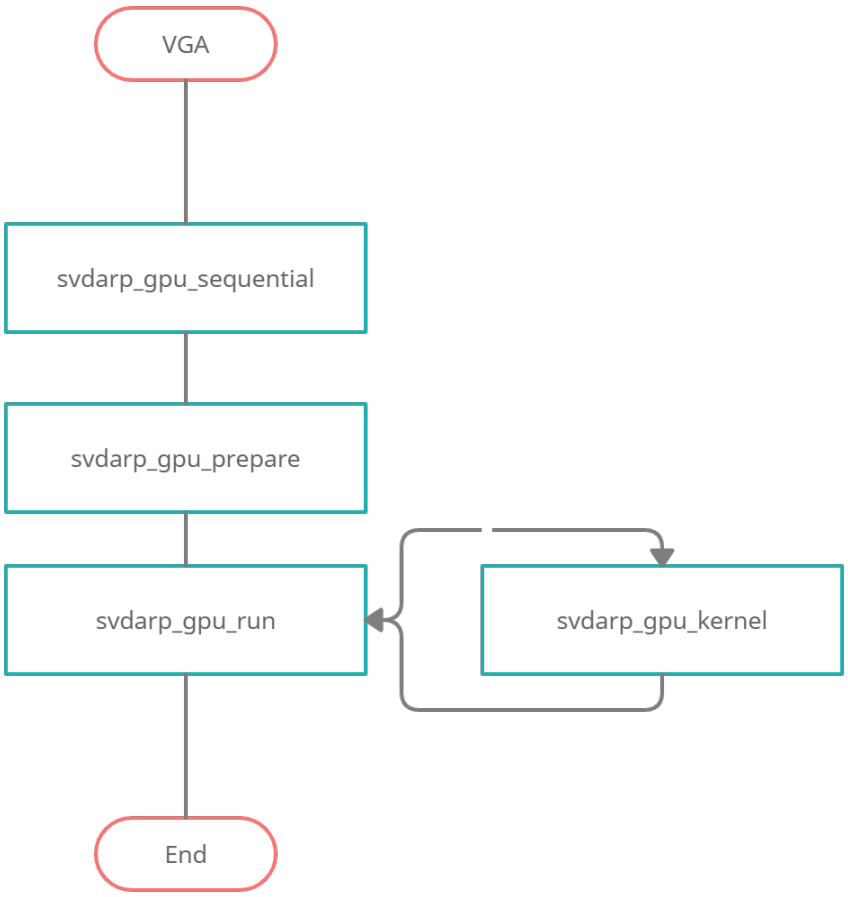

Figure 5.2. Visualization of the main functions used in the GPU parallel algorithm

The sequential search used in the first stage for finding feasible partial solutions at a specified depth is based on the Algorithm 5.1. Instead of saving the vehicle plan as a best solution at the maximum depth tt is modified to save the current vehicle plan when a specified depth is reached and the plan is still feasible.

```
void svdarp_gpu_sequential(InstanceData& instance,
                           std::vector<PlanData>& plan,
                           std::vector<int>& used_nodes,
                           std::vector<PlanData>& partial_solutions,
                           int max_depth
                           int position = 0)
{
    for (int i = 0; i < data.request_nodes.size(); ++i)
    {
        if (used_nodes[i] == -1)
```

```
        {
            plan.nodes[position].node_id = i;
            used_nodes[i] = position;
            bool is_feasible = evaluate(plan, used_nodes, data);

            if (is_feasible)
            {
                if (position == max_depth)
                {
                    svdarp_gpu_store(plan, partial_solutions);
                }
                else {
                    svdarp_gpu_sequential(instance, plan,
                                          used_nodes, stored,
                                          max_depth,
                                          position + 1);
                }
            }
            used_nodes[i] = -1;
        }
    }
}
```

Algorithm 5.4. GPU Sequential search

The function `svadrp_gpu_store` copies the current VehiclePlan `plan` to the collection of partial sollutions. The entire plan is deep copieed and emplaced at the back of the vector.

The `svadrp_gpu_prepare` function determines the maximal count of the partial solutions that can be copied to the GPU at once. The required memory is calculated using `get_memory_requirements` function which takes into consideration the memory requirements specified in Chapter 5. If the memory required for the entire collection of the partial solutions is larger than the available memory on the GPU the number of partial solutions to be copied is decreased and the memory requirements are calculated again until it fits.

```
void svdarp_gpu_prepare(InstanceData& instance,
                        std::vector<PlanData>& partial_solutions)
{
 size_t gpu_available_memory = get_gpu_free_memory();
 size_t chunk_size = partial_solutions.size();
 size_t required_memory = get_memory_requirements(chunk_size,
                                                  block_size);
 void* gpu_instance_data, gpu_partial_solutions;
 while (required_memory > gpu_available_memory);
 {
  chunk_size = decrease_chunk_size(chunk);
  required_memory = get_memory_requirements(chunk_size, block_size);
 }
 cudaMalloc(gpu_partial_solutions,
            chunk_size * sizeof(partial_solutions[0]));
 cudaMalloc(gpu_instance_data, instance_data);
 cudaMemcpy(gpu_instance_data, instance_data, sizeof(instance_data));
```

```
  cudaDeviceSetLimit(cudaLimitStackSize, 32768);

  svdarp_gpu_run(gpu_partial_solutions, partial_solutions,
                 chunk_size, block_size, instance);
}
```

Algorithm 5.5. GPU Preparation

After the memory requirements are settled the memory on the GPU is then allocated for the partial solutions and for the instance data. The instance data is copied over to the GPU. The thread stack size is slightly increased using `cudaDeviceSetLimit` as keeping it at the default value often causes runtime errors.

The function `svdarp_gpu_run` is used to send the data to the GPU and then retrieve complete solutions. It can also divide the collection of the partial solutions into smaller chunks should they not fit in the GPU memory all at once. The partial solutions are copied over to the GPU and the GPU kernel is launched. The best solutions is then retrieved from the GPU using the function `gpu_best_solution`. The function copies the best solutions of the thread blocks back to the host memory. This can occur multiple times until all partial solutions have been explored by the GPU. Then it iterates over all the solutions and finds the one with the best cost. If it is better than the overall best solution it is deep copied and set as the new overall best solution.

```
VehiclePlan svdarp_gpu_run(void* gpu_partial_solutions,
  std::vector<PlanData>& partial_solutions, size_t chunk_size,
  size_t block_size, InstanceData& instance)
{
  size_t processed_count = 0;
  VehiclePlan best_solution;
  best_plan.cost = NUMERIC_MAX;
  void* gpu_solutions = gpu_allocate_solutions(chunk_size, block_size);
  while (processed_count < partial_solutions.size())
  {
    int remaining = partial_solutions.size() - processed_count;
    int copy_size = (remaining < chunk_size) ? remaining : chunk_size;
    cudaMemcpy(gpu_partial_solutions,
               partial_solutions.data()[processed_count], copy_size);
    block_count = ceil(copy_size \ block_size);
    svdarp_gpu_kernel<<<block_count, block_size>>>(instance,
                gpu_partial_solutions, gpu_solutions, copy_size);
    cudaDeviceSynchronize();
    best_solution = gpu_best_solution(gpu_solutions, best_solution);
    processed_count += copy_size;
  }
  return best_solution;
}
```

Algorithm 5.6. GPU Kernel runner

The function `gpu_allocate_solutions` allocates memory on the GPU for storing the best solutions of the thread blocks and the single threads. Pointer to the best solutions of the thread blocks and single threads can either be passed to the kernel as an argument or copied to the GPU memory using *cudaMemcpyToSymbol*.

The `svdarp_gpu_kernel` is the main kernel function launched on the GPU. Each thread initializes it's best solution and the master thread also initializes the best thread block solution. The `tid` represents the index of the partial solution the thread is supposed to explore.

```
__global__ svdarp_gpu_kernel(InstanceData instance,
    PlanData *partial_solutions, PlanData* solutions, size_t chunk_size)
{
    size_t tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (threadIdx.x == 0)
    {
        gpu_init_block_solution(blockIdx.x);
    }
    gpu_init_thread_solution(thread_id);

    if (thread_id < chunk_size)
    {
        gpu_backtrack(instance, tid, partial_solutions);
    }
    __syncthreads();

    if (threadIdx.x == 0)
        gpu_save_block_solution(blockIdx.x, blockDim.x);
}
```

Algorithm 5.7. GPU Kernel

The `gpu_backtrack` function is identical to the serial version of the SVDARP algorithm except the best solution found by the thread is compared and saved to the thread's best solution instead of the global best solution. All functions that are called on the GPU should be marked as either `__global__` or `__device__`. At the end the master thread goes through all the complete solutions found by the threads in the block and chooses the best one as the block's best solution using the `gpu_save_block_solution` function.

Chapter 6
Results

In this chapter the performance of the different approaches to parallelization is measured and compared against the original serial algorithm. Both of the parallel versions of the backtracking algorithm contain tunable parameters which can influence the performance significantly. The impact of each parameter on performance is investigated.

The testing instances used for running the experiments are available in a repository on the faculty's gitlab[1]. Most of the testing scenarios are based on the DARP dataset published in [6]. Slight modifications had to be made to them in order for the datasets to be usable in SVDARP. The scenarios used in this project are made by taking all user requests from a single plan from the published solution. This ensures that a feasible solution exists for given requests.

The experimental results presented in this section were obtained on the RCI cluster[2]. The cluster nodes are equipped with Intel Xeon Silver 4110 CPUs and NVIDIA Tesla V100 GPUs with 32GB of graphic memory. The source code of the project was compiled using GCC version 10.2.0 and NVCC 11.1. The compiler flags used for compilation were `-O2 -march=native`. All times presented here are an average of 3 separate runs with the exact same settings.

6.1 Methods Comparison

This section compares the performance of the three versions of the backtracking algorithm on instances of varying sizes. Table 6.1 contains the execution times of the serial algorithm, the CPU parallel algorithm running on 16 threads and the GPU algorithm using block size of 32. The improvement column is given as the execution time of the serial algorithm divided by the execution time of the parallel algorithm.

As can be seen, the serial version of the algorithm had a superior performance on instances of size up to 18. The CPU parallel version with proper parameter tuning offered the best performance on large instances. On smaller instances it suffers from an overhead of the OpenMP tasks. The GPU parallel version suffers from an overhead required to start the GPU kernel on the smaller instances. On the larger instances it performed better than the serial version, but worse than the CPU parallel version. A visual comparison of a few select instances is provided as Figure 6.1.

[1] https://gitlab.fel.cvut.cz/kulhalu8/cuda-svdarp/-/tree/master/dataset
[2] http://rci.cvut.cz

Nodes	Instance	Serial [s]	CPU [s]	CPU Improvement	GPU [s]	GPU Improvement
10	pr06.v2	0.001	0.051	0.02	0.142	0.01
10	pr15.v11	0.0	0.043	0.0	0.146	0.0
12	pr13.v5	0.001	0.045	0.02	0.143	0.01
12	pr19.v2	0.029	0.054	0.54	0.152	0.19
14	pr03.v1	0.002	0.033	0.06	0.163	0.01
14	pr17.v2	0.096	0.037	2.59	0.149	0.64
16	pr05.v10	0.182	0.035	5.2	0.322	0.57
16	pr17.v3	0.302	0.063	4.79	0.363	0.83
18	pr04.v3	0.021	0.037	0.57	0.182	0.12
18	pr17.v4	5.428	0.658	8.25	3.637	1.49
20	pr02.v5	2.511	0.27	9.3	1.239	2.03
20	pr12.v1	10.829	0.596	18.17	6.46	1.68
22	pr05.v1	0.921	0.12	7.68	0.654	1.41
22	pr20.v5	16.883	2.009	8.4	7.922	2.13
24	pr08.v3	0.249	0.066	3.77	0.163	1.53
24	pr12.v3	514.084	50.799	10.12	128.137	4.01
26	pr01.v3	5.269	1.37	3.85	3.228	1.63
26	pr20.v2	392.612	79.555	4.94	203.657	1.93
28	pr03.v3	33.237	4.901	6.78	12.302	2.7
28	pr09.v6	49.935	15.603	3.2	21.911	2.28
30	pr05.v6	493.031	62.748	7.86	102.608	4.8
30	pr06.v3	52.633	11.027	4.77	31.351	1.68

Table 6.1. Performance of all versions of the algorithm

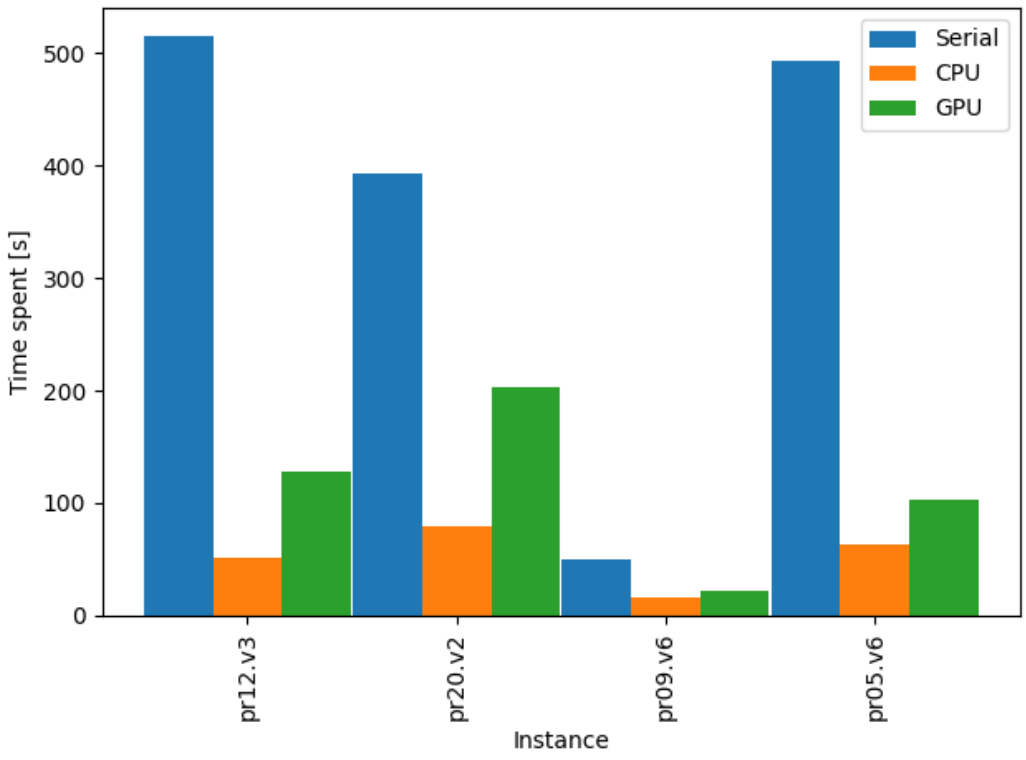

Figure 6.1. Performance of all versions of the algorithm on a few select instances

6.2 Parallel CPU Algorithm Depth

The parallel CPU algorithm uses the parameter `PARALLEL_DEPTH_END` to determine where serialization should occur as explained in Algorithm 5.3. The results were obtained on the instance `pr12.v3` which contains 22 user nodes (11 user requests). The measured results are visualized in Figure 6.2 The detailed results are presented in Table 6.2.

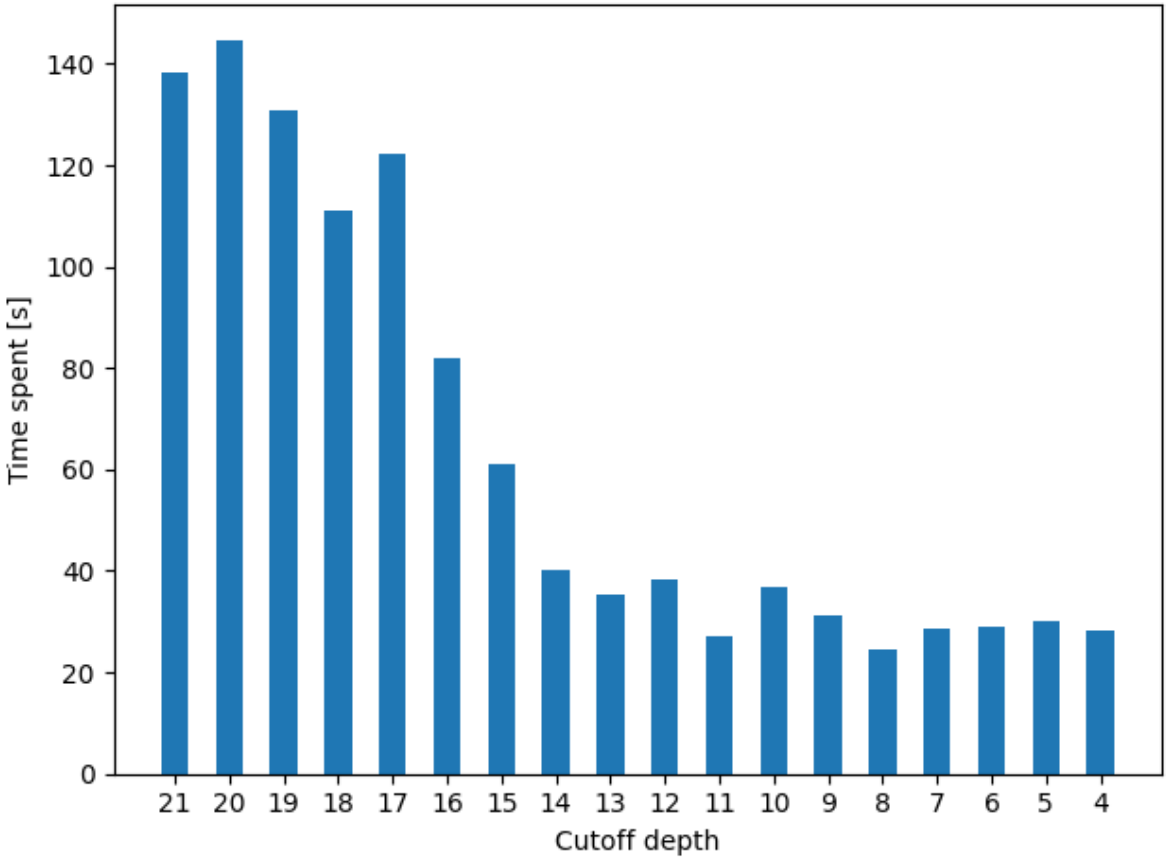

Figure 6.2. CPU depth parameter performance

As can be seen in Table 6.2 the depth at which the serialization occurs influences the performance significantly. The worst performance occured with the parameter set to 20 where the execution time was 144.5 seconds which is 3.56x faster than the serial version of the algorithm. The best performance occured with the parameter set to 8 with an execetuion time of 24.7 seconds which is 20.8x faster than the serial version.

6.3 Parallel CPU Algorithm Thread Count

The parallel CPU algorithm is in theory able to utilize as many cores as possible. However, as shown in [26] the performance of the OpenMP tasks approach usually does not scale linearly with thread count. The results were obtained on the instance `pr12.v3` with `PARALLEL_DEPTH_END` set to 12. The measured results are visualized in Figure 6.3 The detailed results are presented in Table 6.3.

Serialization depth	Run time [s]	Improvement
21	138.4	3.71
20	144.5	3.56
19	130.7	3.93
18	110.9	4.64
17	122.1	4.21
16	81.8	6.28
15	61.2	8.4
14	40.3	12.76
13	35.3	14.56
12	38.4	13.39
11	27.3	18.83
10	36.8	13.97
9	31.2	16.48
8	24.7	20.81
7	28.5	18.04
6	29.1	17.67
5	30.1	17.08
4	28.4	18.1

Table 6.2. Performance of the CPU parallel algorithm depending on the depth of serialization

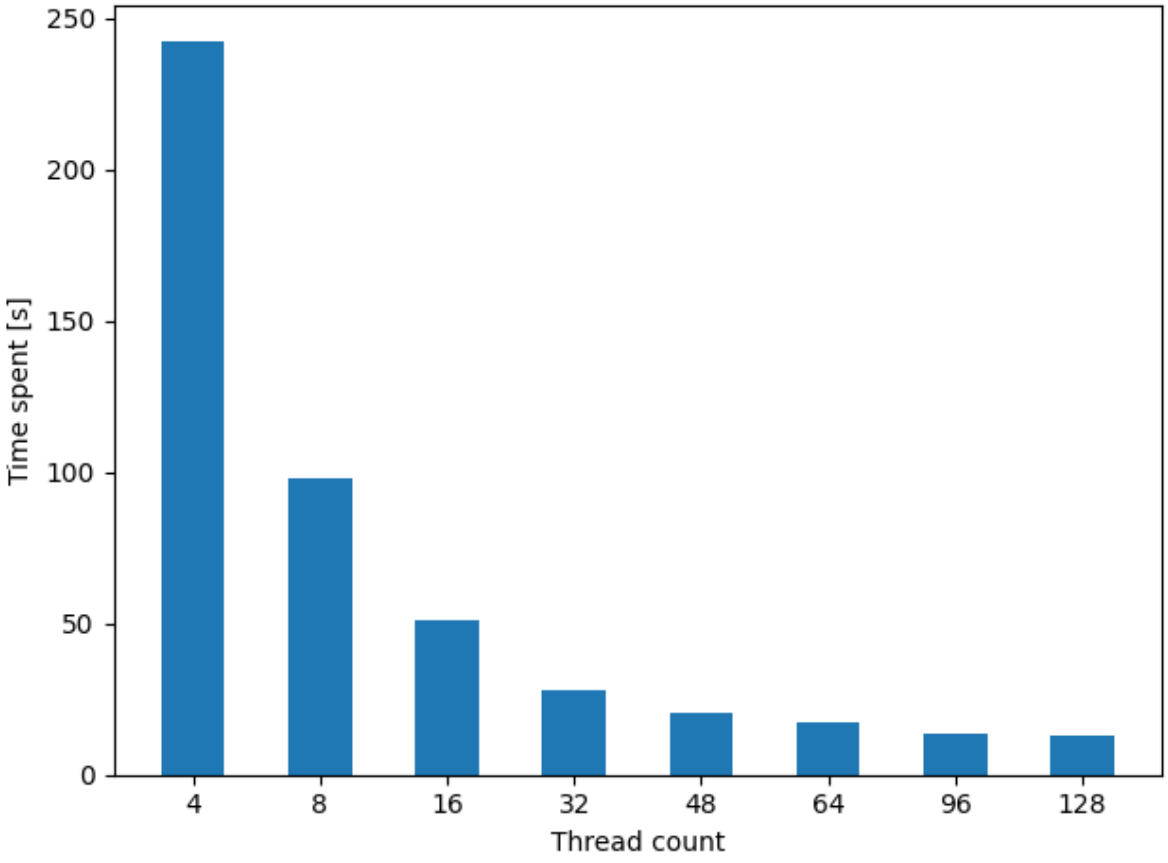

Figure 6.3. CPU thread count performance

As can be seen in Table 6.3 even with just 4 threads the parallel CPU algorithm was 2.12x faster compared to the serial version. With 16 threads, which is the usual thread count of the CPUs in modern desktop computers, the parallel CPU algorithm finished 10.12x faster than the serial version.

Thread count	Run time [s]	Improvement
4	242.2	2.12
8	98.1	5.24
16	50.8	10.12
32	28.1	18.29
48	20.4	25.2
64	17.3	29.72
96	13.5	38.08
128	12.8	40.16

Table 6.3. Performance of the CPU parallel algorithm depending on the number of threads

6.4 Parallel GPU Algorithm Block Size

The block size parameter in Algorithm 5.6 influences the amount of threads in a thread block on the GPU. The results were obtained on the instance `pr12.v3` with `max_depth` set to 12. The measured results are visualized in Figure 6.4 The detailed results are presented in Table 6.4.

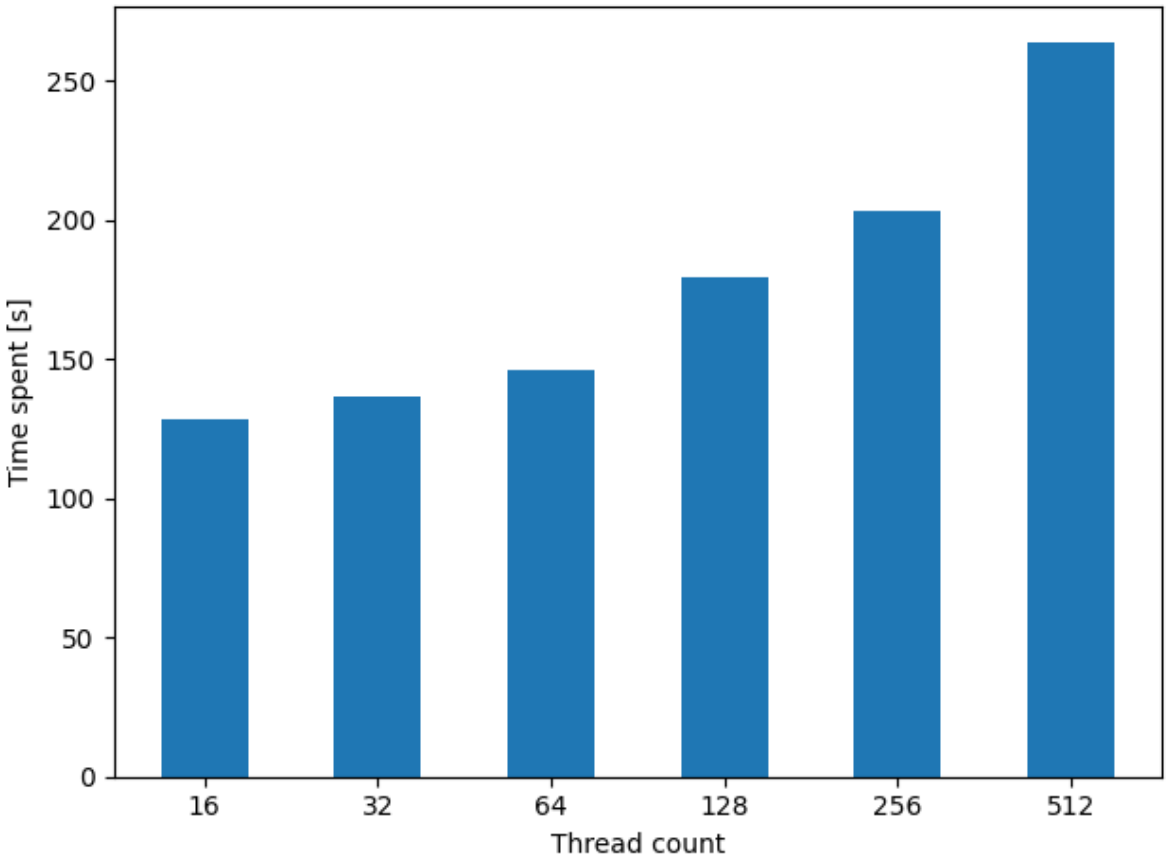

Figure 6.4. GPU threads per block performance

Block Size	Run time [s]	Improvement
16	128.2	4.01
32	136.8	3.76
64	146.2	3.52
128	179.3	2.87
256	203.4	2.53
512	263.6	1.95

Table 6.4. Performance of the GPU parallel algorithm depending on the number of threads in a block

6.5 Parallel GPU Sequential Search Depth

The `max_depth` parameter in Algorithm 5.4 influences the degree of parallelism of the GPU algorithm. A new thread is created for every valid node discovered at `max_depth`. Picking a too low value for the parameter could lead to an underutilization of the GPU as not enough threads will be created. On the other hand a value too high could lead to non-optimal performance due to high memory requirements and therefore a need to split the workload to multiple kernel runs as described in Algorithm 5.6. The results were obtained on the instance `pr12.v3` with block size set to 32. The measured results are visualized in Figure 6.5 The detailed results are presented in Table 6.5.

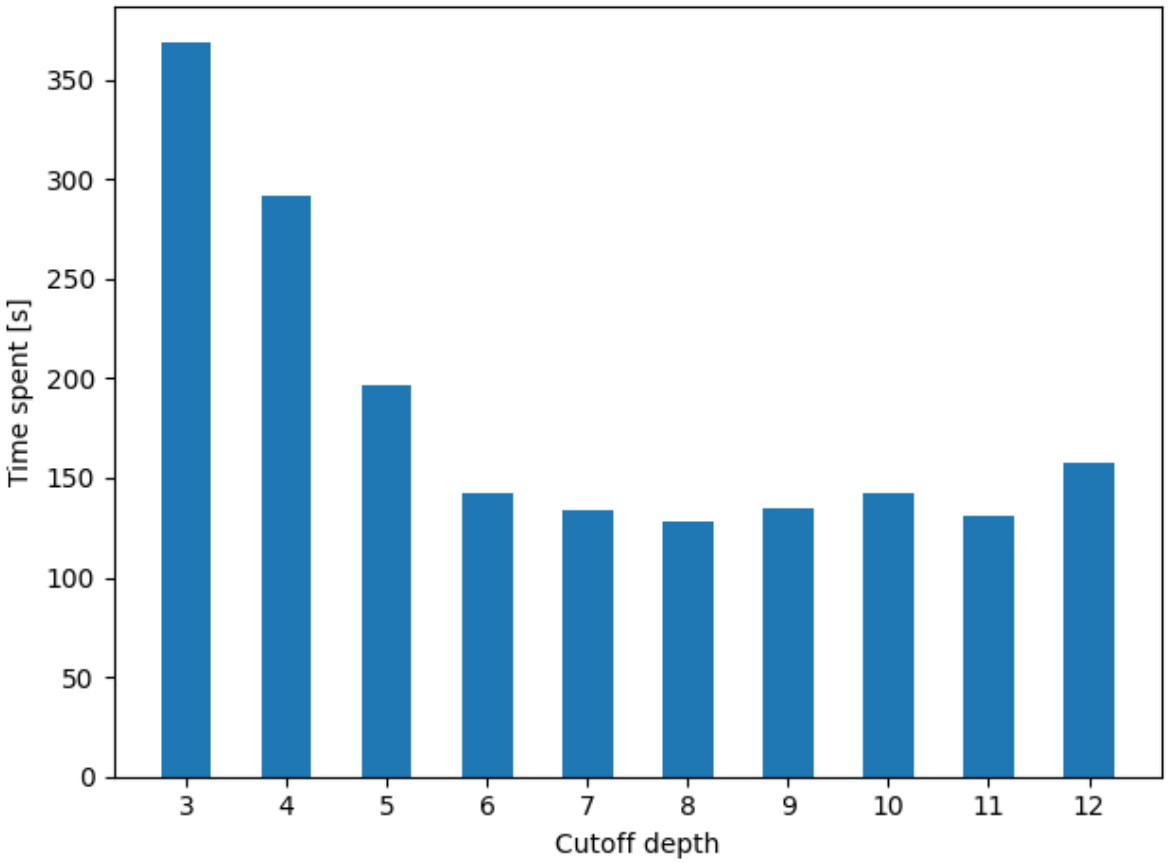

Figure 6.5. GPU sequential depth search performance

Depth Cutoff	Run time [s]	Improvement
3	368.3	1.4
4	292.1	1.76
5	196.7	2.61
6	142.5	3.61
7	133.4	3.85
8	128.2	4.01
9	134.4	3.83
10	142.3	3.61
11	131.2	3.92
12	157.9	3.26

Table 6.5. Performance of the GPU parallel algorithm depending on the sequential search depth

Chapter 7
Conclusion

In this thesis, two methods of parallelization of the backtracking algorithm used for solving the SVDARP were proposed. The parallel GPU version of the algorithm uses CUDA to offload the computations to the graphic card. The solution was implemented in the darp-benchmark project and experimentally evaluated on the RCI cluster.

The implementation proved to be very challenging, as the CUDA framework supports only barebone C++ without the standard library and without the language features of the current C++20 specification. The inability to perform deep copies from the host to the GPU device complicated every step of the implementation, as the data types used in the darp-benchmark project had to be converted to newly proposed flattened data types. Debugging the code running on the GPU device proved to be very tricky, as the runtime errors reported by the CUDA framework were usually very generic and finding the problematic portions of the implemented code was therefore very difficult. It seems the same sentiment is shared by at least some authors [16].

Although results in the work [16] show that speedups achieved using the CUDA version were in some cases comparable to a parallel CPU version run on a 40 thread CPU, such results were not achieved in this thesis. This could be attributed to the nature of the SVDARP, as the problems evaluated in the paper [16] are computationaly much simpler than the SVDARP. The implementation of the SVDARP evaluation function contains a lot of branching, which could be the cause of the lower performance as the divergent branches lead to thread serialization on the GPU [27]. This seems to be supported by the fact that results in Section 6.4 show that the execution time increases with more threads in a block. Overall though the results of the GPU parallel version show some performance improvement at least on larger data instances, performing up to 4 times faster than the serial version on some instances.

The results of the CPU parallel version show that with proper parameter tuning, the performance of the proposed solution could be up to 20 times faster on large instances with lower core count and up to 40 times faster with 128 CPU cores allocated.

An option for further research could be exchanging the evaluation scheme used in the SVDARP algorithm for the recently proposed evaluation scheme published in[28]. The new evaluation scheme was proven to have $\mathcal{O}(n)$ time complexity.

References

[1] M. Chassaing, C. Duhamel, and P. Lacomme. An ELS-based approach with dynamic probabilities management in local search for the Dial-A-Ride Problem. *Engineering Applications of Artificial Intelligence*. 2016, 48 119–133. DOI 10.1016/j.engappai.2015.10.002.

[2] Sophie Parragh, Karl Doerner, and Richard Hartl. Variable neighborhood search for the dial-a-ride problem. *Computers & Operations Research*. 2010, 37 1129–1138. DOI 10.1016/j.cor.2009.10.003.

[3] Masmoudi Mohamed Amine, Manar Hosny, Emrah Demir, and Erwin Pesch. Hybrid adaptive large neighborhood search algorithm for the mixed fleet heterogeneous dial-a-ride problem. *Journal of Heuristics*. 2020, 26 DOI 10.1007/s10732-019-09424-x.

[4] Jean-Francois Cordeau, and Gilbert Laporte. The dial-a-ride problem: models and algorithms. *Annals of Operations Research*. 2007, 153 (1), 29–46. DOI 10.1007/s10479-007-0170-8.

[5] Michal Čáp, and Javier Alonso-Mora. *Multi-Objective Analysis of Ridesharing in Automated Mobility-on-Demand*. 2018.

[6] Jean-Francois Cordeau, and Gilbert Laporte. A tabu search heuristic for the static multi-vehicle dial-a-ride problem. *Transportation Research Part B: Methodological*. 2003, 37 (6), 579–594. DOI 10.1016/S0191-2615(02)00045-0.

[7] Yvan Dumas, Jacques Desrosiers, and Francois Soumis. The pickup and delivery problem with time windows. *European Journal of Operational Research*. 1991, 54 (1), 7–22. DOI 10.1016/0377-2217(91)90319-Q.

[8] Paolo Toth, and Daniele Vigo. *The Vehicle Routing Problem*. 2002. Journal Abbreviation: SIAM Publication Title: SIAM.

[9] Peter van Beek. *Chapter 4 - Backtracking Search Algorithms*. Handbook of Constraint Programming. 2006.
`https://www.sciencedirect.com/science/article/pii/S1574652606800088`.

[10] Richard M. Karp, and Yanjun Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM*. 1993, 40 (3), 765–789. DOI 10.1145/174130.174145.

[11] A. Bruen, and R. Dixon. The n-queens problem. *Discrete Mathematics*. 1975, 12 (4), 393–395. DOI 10.1016/0012-365X(75)90079-5.

[12] Faisal N. Abu-Khzam, Khuzaima Daudjee, Amer E. Mouawad, and Naomi Nishimura. On scalable parallel recursive backtracking. *Journal of Parallel and Distributed Computing*. 2015, 84 65–75. DOI 10.1016/j.jpdc.2015.07.006.

[13] Kamil Rocki, and Reiji Suda. *Parallel Minimax Tree Searching on GPU*. In: Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, eds. *Parallel Processing and Applied Mathematics*. Berlin, Heidelberg: Springer, 2010. 449–456. ISBN 978-3-642-14390-8.

[14] Tiago Pessoa, Jan Gmys, Nouredine Melab, Francisco de Carvalho-Junior, and Daniel Tuyttens. *A GPU-Based Backtracking Algorithm for Permutation Combinatorial Problems.* In: 2016. 310–324. ISBN 978-3-319-49582-8.

[15] Faisal N. Abu-Khzam, Mohamad A. Rizk, Deema A. Abdallah, and Nagiza F. Samatova. *The Buffered Work-Pool Approach for Search-Tree Based Optimization Algorithms.* In: Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, eds. *Parallel Processing and Applied Mathematics*. Berlin, Heidelberg: Springer, 2008. 170–179. ISBN 978-3-540-68111-3.

[16] Tiago Carneiro Pessoa, Jan Gmys, Francisco Heron de Carvalho Júnior, Nouredine Melab, and Daniel Tuyttens. GPU-accelerated backtracking using CUDA Dynamic Parallelism. *Concurrency and Computation: Practice and Experience*. 2018, 30 (9), e4374. DOI https://doi.org/10.1002/cpe.4374. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4374.

[17] Peter Zhang, Eric Holk, John Matty, Samantha Misurda, Marcin Zalewski, Jonathan Chu, Scott McMillan, and Andrew Lumsdaine. *Dynamic parallelism for simple and efficient GPU graph algorithms.* In: *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. New York, NY, USA: Association for Computing Machinery, 2015. 1–4. ISBN 978-1-4503-4001-4. `https://doi.org/10.1145/2833179.2833189`.

[18] Max Plauth, Frank Feinbube, Frank Schlegel, and Andreas Polze. *Using Dynamic Parallelism for Fine-Grained, Irregular Workloads: A Case Study of the N-Queens Problem.* In: *2015 Third International Symposium on Computing and Networking (CANDAR)*. 2015. 404–407. ISSN: 2379-1896.

[19] Jacques Desrosiers, Yvan Dumas, and F. Soumis. A Dynamic Programming Solution of the Large-Scale Single-Vehicle Dial-A-Ride Problem with Time Windows. *American Journal of Mathematical and Management Sciences*. 1986, 6 DOI 10.1080/01966324.1986.10737198.

[20] Harilaos N. Psaraftis. Analysis of an O(N2) heuristic for the single vehicle many-to-many Euclidean dial-a-ride problem. *Transportation Research Part B: Methodological*. 1983, 17 (2), 133–145. DOI 10.1016/0191-2615(83)90041-3.

[21] Shen Lin. Computer solutions of the traveling salesman problem. *The Bell System Technical Journal*. 1965, 44 (10), 2245–2269. DOI 10.1002/j.1538-7305.1965.tb04146.x. Conference Name: The Bell System Technical Journal.

[22] S. Lin, and B. W. Kernighan. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research*. 1973, 21 (2), 498–516. Publisher: INFORMS.

[23] Lauri Häme. An adaptive insertion algorithm for the single-vehicle dial-a-ride problem with narrow time windows. *European Journal of Operational Research*. 2011, 209 (1), 11–22. DOI 10.1016/j.ejor.2010.08.021.

[24] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU graph traversal. *ACM SIGPLAN Notices*. 2012, 47 (8), 117–128. DOI 10.1145/2370036.2145832.

[25] Eduard Ayguade, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*. 2009, 20 (3), 404–418. DOI 10.1109/TPDS.2008.105. Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[26] Haoqiang Jin, Dennis Jespersen, Piyush Mehrotra, Rupak Biswas, Lei Huang, and Barbara Chapman. High performance computing using MPI and OpenMP on multi-core parallel systems. *Parallel Computing*. 2011, 37 (9), 562–575. DOI 10.1016/j.parco.2011.02.002.

[27] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel Computing Experiences with CUDA. *IEEE Micro*. 2008, 28 (4), 13–27. DOI 10.1109/MM.2008.57. Conference Name: IEEE Micro.

[28] Murat Firat, and Gerhard J. Woeginger. Analysis of the dial-a-ride problem of Hunsaker and Savelsbergh. *Operations Research Letters*. 2011, 39 (1), 32–35. DOI 10.1016/j.orl.2010.11.004.

Appendix A
Thesis Specification

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kulhánek** Jméno: **Lukáš** Osobní číslo: **474407**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Software**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Paralelizace backtracking algoritmu pro Single-vehicle DARP s pomocí grafické karty

Název bakalářské práce anglicky:

GPU Parallelization of the Backtracking Algorithm for Single-vehicle DARP

Pokyny pro vypracování:

Seznam doporučené literatury:

[1] T. Carneiro Pessoa, J. Gmys, F. H. de C. Júnior, N. Melab, and D. Tuyttens, 'GPU-accelerated backtracking using CUDA Dynamic Parallelism', Concurrency and Computation: Practice and Experience, vol. 30, no. 9, p. e4374, 2018, doi: 10.1002/cpe.4374.
[2] R. Finkel and U. Manber, 'DIB - a distributed implementation of backtracking', ACM Trans. Program. Lang. Syst., vol. 9, no. 2, pp. 235–256, Mar. 1987, doi: 10.1145/22719.24067.
[3] V. N. Rao and V. Kumar, 'On the efficiency of parallel backtracking', IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 4, pp. 427–437, Apr. 1993, doi: 10.1109/71.219757.
[4] E. Speckenmeyer, B. Monien, and O. Vornberger, 'Superlinear speedup for parallel backtracking', in Supercomputing, Berlin, Heidelberg, 1988, pp. 985–993, doi: 10.1007/3-540-18991-2_58.
[5] J. Jenkins, I. Arkatkar, J. D. Owens, A. Choudhary, and N. F. Samatova, "Lessons Learned from Exploring the Backtracking Paradigm on the GPU," in Euro-Par 2011 Parallel Processing, Berlin, Heidelberg, 2011, pp. 425–437, doi: 10.1007/978-3-642-23397-5_42.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. David Fiedler, centrum umělé inteligence FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **04.03.2021** Termín odevzdání bakalářské práce: _____

Platnost zadání bakalářské práce: **19.02.2023**

Ing. David Fiedler podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

