Train Re-scheduling
A massively Parallel Approach Using CUDA

Anton Petersson

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden
This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author:
Anton Petersson
E-mail: anpk12@student.bth.se

University advisor:
Dr. Johanna Törnquist Krasemann
Dept. Computer Science & Engineering

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57
ABSTRACT

Context. Strategic release planning (sometimes referred to as road-mapping) is an important phase of the requirements engineering process performed at product level. It is concerned with selection and assignment of requirements in sequences of releases such that important technical and resource constraints are fulfilled.

Objectives. In this study we investigate which strategic release planning models have been proposed, their degree of empirical validation, their factors for requirements selection, and whether they are intended for a bespoke or market-driven requirements engineering context.

Methods. In this systematic review a number of article sources are used, including Compendex, Inspec, IEEE Xplore, ACM Digital Library, and Springer Link. Studies are selected after reading titles and abstracts to decide whether the articles are peer reviewed, and relevant to the subject.

Results. 24 strategic release planning models are found and mapped in relation to each other, and a taxonomy of requirements selection factors is constructed.

Conclusions. We conclude that many models are related to each other and use similar techniques to address the release planning problem. We also conclude that several requirement selection factors are covered in the different models, but that many methods fail to address factors such as stakeholder value or internal value. Moreover, we conclude that there is a need for further empirical validation of the models in full scale industry trials.

Keywords: 3-4 keywords, maximum 2 of these from the title, starts 1 line below the abstract.
I thank Johanna Törnquist Krasemann at BTH for all the advice, help, explanations and encouragement. Your support has really helped me complete this thesis. Thank you.

I have also had very good discussions with Håkan Grahn at BTH. You really helped me decide on an initial design of the parallel algorithm. Thank you.

Henrik Andersson at NTNU also offered very useful advice. In addition, he offered me a place to work on my thesis at NTNU. Thank you.

I also thank Rolf Olstad at ZXY Sport Tracking / ChyronHego for allowing me to work on my thesis in their office in Trondheim. I do not think I could have made the same progress outside of this environment.

My brother Simon Sjungare read some very early drafts of my thesis, offered useful advice and encouraged me when I was stressed out. Thank you.
List of Figures

1.1 Train B must wait on Train A to arrive since segment 2 only has a single track. .......................................................... 2
1.2 Solution space as a tree. .................................................. 3
3.1 Software overview. ......................................................... 8
3.2 Segment allocation lists. .................................................. 10
3.3 Overview of the GPU-based approach. .............................. 13
3.4 Candidate lists. Every thread has its own candidate list. Every candidate list starts with a counter for the number of candidates currently in the list. The candidates are simply indices into the event definition list and into the event lists. .................... 16
3.5 Segment definitions and allocation lists. Every thread has its own set of segment allocation lists, one per segment. The read-only properties of segments are stored in an array shared between all threads. In this example, the allocation lists currently hold one allocated event each. ........................................... 17
3.6 Finding the starting points. .............................................. 18
4.1 Cost of best solutions found for scenarios 1 and 2 with 115 as branching depth. ......................................................... 26
4.2 Number of deadlocks with increasing number of threads, branching at depth 10. .......................................................... 27
4.3 Kernel execution times for scenario 1 with varying number of threads and block dimensions. ........................................... 28
4.4 Total execution times vs kernel execution times for scenarios 1 through 16, with 4096 threads. ................................. 29
4.5 Performance comparison between the parallel algorithm and the sequential algorithm using 22528 threads and a branching depth of 50. ................................................................. 30
4.6 Performance comparison between the parallel algorithm and the sequential algorithm using 22528 threads and a branching depth of 100. ................................................................. 31
4.7 Performance comparison between the parallel algorithm and the sequential algorithm using 22528 threads and a branching depth of 150.

4.8 Performance comparison between the parallel algorithm and the sequential algorithm using 22528 threads and a branching depth of 200.

4.9 Performance comparison based on kernel performance without overhead. Branching depth 200.
List of Tables

3.1 Disturbance scenarios. ........................................ 21
3.2 Test system used in the experiments. ...................... 22
4.1 Costs of solutions found for scenario 1 when branching
    depth 10. ...................................................... 25
4.2 Costs of solutions found for scenario 2 when branching at depth 10. 25
Abstract

1 Introduction
   1.1 Train re-scheduling .................................... 1
   1.2 Algorithmic approach ................................... 3
   1.3 Using a GPU ............................................ 3
   1.4 Our GPU approach ...................................... 5
   1.5 Research questions .................................... 5

2 Related Work .................................................. 6

3 Method ................................................................... 8
   3.1 Dataset ..................................................... 8
   3.2 Sequential algorithm .................................... 10
       3.2.1 Chunk scheduling .................................. 12
   3.3 GPU algorithm ........................................... 12
       3.3.1 Memory layout of the GPU implementation ....... 13
       3.3.2 Preparing the input to the GPU ................. 18
       3.3.3 CUDA-C Kernel .................................... 20
   3.4 Disturbance scenarios .................................... 21
   3.5 Experimental setup ...................................... 21
   3.6 Validity threats .......................................... 22

4 Results ................................................................ 24
   4.1 Experiment 1: Number of deadlocked branches ....... 24
   4.2 Experiment 2: Performance and scalability .......... 24
   4.3 Experiment 3: Data transfer overhead ................. 28
   4.4 Experiment 4: Performance comparison between sequential and parallel algorithm ................. 29

5 Analysis ................................................................ 35

6 Conclusions and Future Work ............................... 39
Chapter 1

Introduction

This thesis focuses on using a graphics processing unit (GPU) to solve the train re-scheduling problem. We first explain the train re-scheduling problem in detail. Then we discuss the potential benefits and difficulties of using a GPU to solve it.

1.1 Train re-scheduling

Railway networks can be divided into segments. A segment is either a station or not. There may be one or more (non-station) segments between stations. Each segment, regardless of type, can have one or more tracks. Each segment track can only accommodate one train at a time on any given track.

We focus on networks where only the stations have multiple tracks, although the algorithm that we develop should support multi-track non-station segments as well.

Railway networks typically have multiple trains, each with its own timetable. Timetables are a sequence of events. An event is associated with a train and a segment, has an arrival time, departure time, and a designated track. An event describes when a train is going to arrive to a segment, what track it will use and when it is going to depart. The events of a train must be executed in order. Timetables are created with the capacity of the network in mind in a way that avoids deadlocks.

In this type of single-tracked network, trains traveling in opposite directions can only meet at stations. This leads to situations in which one train traveling west becomes delayed at a multi-track station because the only track going west is occupied by a train traveling east. The west-bound train must then wait for the east-bound train to reach the station before it can start moving. This is illustrated in figure 1.1, where Train B is traveling west and must wait for Train A at segment 3.

The original timetables keep situations like these to a minimum under normal conditions. That is, waiting times are generally short and few. However, in the case of a traffic disturbance, such situations will start to happen more frequently. A disturbance can be anything that causes one or more trains to diverge from their timetables. For instance, a train has some engine failure and has to proceed.
Figure 1.1: Train B must wait on Train A to arrive since segment 2 only has a single track.

with decreased speed. Or a train departs later than planned from a station. We limit the scope to disturbances where one train leaves a segment later than planned. That segment can be a a station segment or non-station segment.

The effects of a disturbance can be controlled by re-scheduling the traffic. The network, or infrastructure, consists of a set of railway segments $S$. Some of the segments are station segments. All segments $s \in S$ have one or more tracks. Each train $i \in T$, where $T$ is the set of all trains, has a predefined set of events. So each train $i \in T$ has its own set of events $E_i$, which is a subset of all the events in the scenario, $E^\text{all}$. Each train $i \in T$ must receive a track and a time slot for every segment $s$ that it travels through. The re-scheduling problem is about finding updated arrival and departure times and tracks for all events $k \in E^\text{all}$ so that

1. the internal order of events for every train is respected, and
2. there are no conflicts.

This can be seen as an instance of the no-wait Job Shop Scheduling problem (no-wait JSS) [13]. The segments can be thought of as machines capable of performing jobs. Each job (train) has to be scheduled on each of the segments in order. A train must at all times be scheduled on a segment (no-wait).

A solution to the re-scheduling problem can be seen as an ordered sequence in which to prioritize and schedule events. The solution space, i.e. the set of all solutions, can then be thought of as a tree in which each node represents an event\(^1\). See figure 1.2 for an illustration of such a tree. In the figure, yellow nodes represent a selected solution.

The topic of this thesis is about finding ways to solve this problem using GPUs, which we get to, and motivate, shortly.

We use a limited model of the problem and only look at the originally planned departure and arrival time, minimum running time and preferred track number. We do not consider platform lengths, train acceleration times etc. In our model, trains are never allowed to leave a segment before the timetable. Trains can only

\(^1\)With chunk allocations, as we describe in section 3.2.1, each node represents a sequence of events from the same train.
catch up if they are already delayed, never build up a buffer by running ahead of schedule.

1.2 Algorithmic approach

There are known approaches to this problem that can find the optimal solutions [17]. But even an optimal set of timetables will be of little help if it takes too long to find. If we use an hour to find the solution, how should we manage the train traffic during that hour? With a large time horizon, many trains and a large network, the set of solutions is large and the task complex. An algorithm running on a powerful computer can potentially generate new timetables faster than a human. In fact, there are a number of such algorithms already [12][9].

Reducing the effects of train disturbances is a focus of many governments, since train travel and transport is seen as a green mode of transportation [3]. In 2005, road transports accounted for 92% of passenger transports within the EU, which was seen as a danger by the European commission [6]. In the same year, only 17% of freight transport in the EU happened by rail. The European Railway Agency (EPA) is working to improve the competitiveness of railway compared to other modes of transportation, calling it "the most environment-friendly mode of transportation" [8].

1.3 Using a GPU

Rendering of computer graphics is inherently a highly data parallel process. Vertex positions and other attributes can be independently computed. The color
(and potentially other attributes) of each pixel can be independently computed. These tasks have influenced the evolution of GPUs greatly. GPUs are designed to perform the same computations on multiple data, in parallel.

Modern GPUs provide a high number of streaming processors (SPs) organized into streaming multiprocessors (SMs) and high memory bandwidth [10]. The programming model enabled by the release of the Compute Unified Device Architecture (CUDA) development toolkit makes it possible to use this hardware for solving generic computational problems in addition to rendering computer graphics [5]. This type of use is often referred to as general purpose GPU computing (GPGPU) or heterogeneous computing.

In many cases, GPUs can outperform CPUs [10, 2]. The main challenge is to formulate the problem in a way so that we can take advantage of the parallel nature of modern GPUs.

Parallel algorithms are needed in order to utilize the full processing capability of GPUs. Running a sequential algorithm on a GPU would be pointless from a performance perspective.

The CUDA toolkit includes a special language for expressing parallel algorithms, CUDA-C. It is basically C with some extensions and allows one to write code to be executed on the host (CPU) and code to be executed on the device (typically a GPU). The GPU code is expressed as kernels. These are special functions that will be executed in parallel by multiple threads on the GPU. Threads can use their thread indices to derive pointers or offsets to different parts of the input and output data structures, enabling multiple data elements to be processed in parallel.

The threads are organized into blocks. Each block can have a number of threads arranged in one, two or three dimensions. The maximum number of threads in one block is limited by the CUDA device used. Threads in a block can collaborate using shared memory and synchronization. But there are device-specific limitations on the amount of shared memory per block and registers per block. The blocks are themselves organized into a grid. Grids can be multidimensional, like blocks.

CUDA-C also includes syntax for invoking kernels and specifying kernel launch parameters. The launch parameters control the number of threads and blocks.

There are many important trade-offs to consider when developing heterogeneous applications using CUDA and we will not cover them here.

"Achieving a scalable high-performance code that uses hardware resources efficiently is still a difficult task that can take months and years to master" [2].

For a thorough introduction to heterogeneous computing using CUDA, see for instance [2].
Chapter 1. Introduction

1.4 Our GPU approach

As mentioned, the solution space can be seen as a tree, where each node is a sequence of events from a train being scheduled. We create a massively parallel GPU algorithm for exploring multiple branches of this tree in parallel. This selection of granularity has the advantage that branches can be explored independently from other branches, which means no synchronization among threads is needed. However, the fact that the threads in our algorithm work independently from each other is also a problem, because it makes it difficult to use shared memory. We instead keep almost all data structures entirely in the slower global memory of the device.

The parallel GPU algorithm that we present in this thesis is not a complete rescheduling approach. It is only a building block that can explore multiple branches in parallel. A high-level algorithm is needed on top to select what branches to explore and to systematically do that until a set time limit is reached. Developing such a high-level algorithm is out of scope for this thesis. Therefore we do not attempt any systematic comparison between the solutions found by the parallel algorithm and the ones found by the sequential one.

Our algorithm does not make use of the CPU once the CUDA kernel has been launched. Instead, the CPU is left idle while the CUDA kernel explores the solution tree.

1.5 Research questions

The problem of optimizing schedules for trains on single-tracked railway lines can be described as a no-wait job shop scheduling problem and as a combinatorial optimization problem. It is not feasible to perform an exhaustive search. Branch-and-bound algorithms have been developed for this problem. Both sequential and parallel. But no parallel GPU approach exist in the literature. This study investigates the potential of using GPUs, through the CUDA toolkit, to enhance the re-scheduling capabilities of a sequential algorithm.

We formulate the following questions:

1. How does the performance of the parallel approach scale with an increased number of threads?

2. How does the parallel algorithm perform compared with the sequential algorithm, in terms of nodes visited per time unit?

We mean computational efficiency or speed when we say "performance" in this context. We are not investigating the optimality of the solutions generated by the algorithms.
Chapter 2

Related Work

The train re-scheduling problem has received a lot of attention from researchers. Surveys of the contributions are available in [11], [7] and more recently [3]. The latter one categorizes train re-scheduling algorithms as either macroscopic or microscopic. Macroscopic algorithms work on a high level and considers stations as nodes in a graph but does not take block sections, signals or platforms into account, while microscopic algorithms do. Our approach can be considered somewhere in between macroscopic and microscopic.

Rescheduling of train traffic can be viewed as a combinatorial optimization problem. Many researchers use commercial optimization software and Mixed Integer Linear Programming (MILP) to solve disturbance scenarios to optimality. Four optimization strategies using based on MILP are presented and evaluated in [17]. The problem with MILP approaches is that it often takes too much time to solve the problem to optimality, or to prove optimality. So some researchers turn to optimization algorithms to try to find good-enough solutions in a short amount of time.

Branch-and-bound (BB) algorithms can be used to find optimal solutions without exploring the entire set of solutions. Branch-and-bound algorithms organize the set of possible solutions into a tree structure. This tree structure is then explored starting from the root node. Each child node represents a partial solution and the algorithm guides the exploration using upper and lower bounds. This way many solutions can be discarded.

A greedy BB algorithm is presented in [12]. This algorithm quickly finds valid schedules by using a depth-first search. When conflicts arise, it uses a naive cost estimation function to prioritize trains. Once it has found an initial solution it enters a phase where it backtracks in order to find better solutions within a given time limit. In our parallel GPU-based algorithm every thread executes a simplified version of this algorithm so that multiple branches can be explored in parallel. An important distinction is also that our GPU-based algorithm uses a very simple heuristic when prioritizing trains. And when a thread runs into a deadlocked solution branch, we abort it, instead of trying to backtrack to search for a non-deadlocked branch. This is because we want to limit thread divergence.

A parallel algorithm for multicore shared memory CPUs is presented in [9].
Chapter 2. Related Work

This algorithm uses multiple threads that share information of current upper bounds and best solutions found so far using a synchronized white board. The algorithm is shown to cover a larger part of the search tree. For five out of 20 scenarios, it finds significantly better solutions than a sequential algorithm. The parallel GPU-algorithm that we present in this thesis is different. We use up to 22528 threads but each thread works independently from the others, sharing no information and doing no synchronization.

A GPU accelerated branch-and-bound algorithm is presented in [15]. This algorithm uses a pool of GPU-threads to compute the lower bounds while the CPU performs elimination, selection and branching. They claim to achieve speedup factors of up to 100x when compared to a sequential implementation. An improved algorithm with reduced thread divergence is presented in [4]. These results contribute to the relevance of investigating the possibility of using GPUs to search for solutions to the train re-scheduling problem. But our GPU-based algorithm is not a BB algorithm.

Another branch-and-bound algorithm taking advantage of GPU processing is presented in [1]. This algorithm is tailored towards the knapsack problem. This paper also emphasizes that GPUs are most efficient when working on data parallel problems but also suggests that they can be utilized for task parallelism as well. Our parallel algorithm is difficult to categorize, but we consider it data parallel. Every thread runs the same algorithm but using different data.

Since the railway disturbance problem requires a new solution to be found in a short amount of time, it is reasonable to use heuristics in the search and focus on finding a near optimal solution. Such methods are called meta-heuristics. One local search meta-heuristics (LSMs) method adapted for GPU processing is presented in [14] with speedups of up to 80x when compared to sequential CPU implementations. Notice that LSMs differ from branch and bound algorithms in that they do not necessarily find the optimal solution, while branch and bound does (unless aborted).

A survey on GPU algorithms focused on routing problems is available in [16].

To our knowledge, there is no published work on how to take advantage of GPU-processing for the train re-scheduling problem, so this is the knowledge gap that we identify and target with this work. We also want to point out that the goal is not to make a faster version of a sequential algorithm, but rather to find ways of using the extra processing power to find better solutions. Of course, being able to cover larger parts of the search tree will be an important building block.
Chapter 3

Method

We develop a sequential algorithm in C++ and a parallel version of the same algorithm in CUDA-C. See figure 3.1 for an overview of the software that we develop. This allows us to study the potential of GPU-based algorithms for the train re-scheduling problem. We construct 16 scenarios with disturbances. In all scenarios one train becomes delayed by 15 to 45 minutes at a railway segment (station or not). The problem then is to find a new schedule adapted to the new situation. Trains that were planned to meet may need to meet at other locations. Then we run a series of benchmark experiments using both the sequential and the parallel algorithms to collect data to answer our research questions.

In this chapter we describe, in detail, the dataset, the sequential algorithm, the parallel algorithm, the data structures involved and the disturbance scenarios.

Figure 3.1: Software overview.

3.1 Dataset

We have exclusively tested our algorithms using variations of a single dataset. This dataset holds the original schedule for 103 trains over one day, May 28 2014, on a subset of Malmöbanan. Malmöbanan is a part of the Swedish railway network. It stretches from Narvik in Norway to Luleå in Sweden. The subset used in this
thesis is from Riksgränsen to Luleå. The dataset is defined by two files: one for the infrastructure and one for the timetable.

The infrastructure file defines all segments. The parameters that we use are:

- Type of segment. Station (LOC) or network (NWK).
- Acronym for the segment. Only used for debugging and validation.
- Name (only for station segments). Only used for debugging and validation.
- Number of tracks.

There are some more parameters that we do not use.

The timetable file describes trains and their events. Every train is defined in terms of id and a list of events, and some additional meta data that we do not use. The type of event is TSP for events that concern LOC segments and TMV for events that concern NWK segments.

In our simplified model of the problem, TSP events have the following properties:

1. Segment index
2. Train id
3. Departure time
4. Arrival time
5. Minimum dwell time
6. Planned stop (yes or no)
7. Originally planned track

TMV events have the following properties:

1. Segment index
2. Train id
3. End time
4. Start time
5. Minimum runtime
6. Preferred track

In total, this dataset has 4245 events, 103 trains and 162 segments. We have tested our algorithms with simplifications and slight variations of this dataset during the development. The disturbance scenarios can be seen as modifications to the dataset, and therefore slightly different datasets. The scenarios used in the experiments are described in section 3.4.
3.2 Sequential algorithm

The sequential algorithm that we have implemented is inspired by the one presented in [12]. We consider this algorithm the state of the art sequential algorithm.

The trains with their events are read into a vector. All trains have a vector each for their respective events. The events are stored in chronological order. Care is taken to make sure that all of the event properties are allocated next to each other in memory in order to ensure a high cache efficiency. This is done by declaring the vector to hold instances of the Event class and not pointers to it.

We use a candidate list to store candidate events. These are events that might be selected for scheduling. The candidate list is kept sorted based on an optimistically estimated earliest departure, without taking the segment allocation lists into account.

We read the segments into a vector. Each segment gets one allocation list per track. The allocation lists are dynamically allocated vectors where every element is a pointer to an event. In the initial condition the allocation lists are empty. As events get scheduled, they are inserted at the end of these allocation lists. The previously inserted events represent timeslots that are already allocated to trains. See figure 3.2.

```cpp
class Segment {
    vector<Event *> allocs[numTracks];
};
```

![Figure 3.2: Segment allocation lists.](image)

All events are initially waiting. One event is scheduled at a time. If an event \( k \) is successfully scheduled and added to the associated segment allocation list, we remove it from the candidate list and mark it active. The next event \( k + 1 \) for the associated train is added to the candidate list. If the train does not have more events, we mark the just scheduled event \( k \) executed.
When a new event $k$ is being scheduled on a segment, we perform a linear search to find the track with the earliest possible start time. When the best track $u^*$ is identified together with the earliest possible start time $b^*_{\text{track}}$ for that track, we add the event to the track’s allocation list. The start time $b_k$ for event $k$ is set to $\max(b^*_{\text{track}}, b^*_{\text{planned}})$, where $b^*_{\text{planned}}$ is the originally planned start time of event $k$. We do this for all segments, regardless of whether they are station segments or not. We also set the state of $k$ to active.

If $k$ is not the first event of a train, event $k - 1$ is expected to be in the active state. We set the stop time $e_{k-1}$ for event $k - 1$ to $b_k$ and the state of $k - 1$ to executed.

We perform a simple conflict and deadlock analysis as part of searching for the best track. When evaluating a track $u$, we iterate over the events in the associated allocation list. Any executed events that we encounter only affects the earliest possible start time $b^*_{\text{track}}$ for the track. The state of the track is still treated as available. When we encounter an active event, we will come to one of three conclusions for the track:

1. Deadlocked
2. Conflicted
3. Delayed

We consider two trains deadlocked if the current event is traveling in the the opposite direction to the active event already in the track allocation list. In that situation, even if we do not make the new allocation, the two trains would have to wait indefinitely for the next segment to become available, unless one of them could select another track.

They are conflicted if both trains are traveling in the same direction but try to claim an overlapping time slot on the same track. Either train can be prioritized. But the second train can not allocate the segment before the first train has become executed, i.e. been scheduled on its next segment, or run out of events.

They are delayed if both trains are traveling in the same direction and the claimed time slots do not overlap. Delayed and conflicted both mean that the current event can not be scheduled yet on that particular track.

In the search for the best track, tracks that lead to two trains being delayed or available are prioritized, but only if they also allow for a better start time for the current event. Only if the best track at the end of the search is available do we commit the allocation, i.e. add it to an allocation list, set the start time and mark it active.

If the best track is deadlocked, a more thorough deadlock analysis is performed. It iterates over all events in the allocation lists of all tracks of the current segment. For each of those events, it looks at the next event of the same train and searches for the best track for that event. If any of those searches finds a track to use that
does not cause a deadlock, the deadlock analysis returns false. The allocation is then delayed so that another candidate event can be investigated instead.

3.2.1 Chunk scheduling

Instead of strictly adhering to the scheduling order defined by the candidate list, we often schedule multiple events from the same train, even if only the first of those are in the candidate list. We do this because the infrastructure often has multiple consecutive single-track segments connecting the multi-track segments. We argue that these single-track segments have to be allocated in sequence for trains. We think that allocating them one at a time only increases the risk for deadlocks, requiring more frequent backtracking. Also, before discovering a deadlock, we might have already scheduled a set of unrelated events on other segments, that would also have to be reversed.

So, we consider it a reasonable simplification to always keep allocating events from the same train until an event on a multi-track segment can be allocated, or the train runs out of events. We call these sequences of single-track events chunk allocations. If we fail to complete a chunk, i.e. one of the events in the chunk cannot be scheduled, we abort the entire chunk by undoing the already scheduled events one by one in reversed order.

Undoing an event includes resetting the state to waiting and removing it from a track allocation list of the segment that the event is associated with. Unless the event is the first event for a train, the state of the previous event will be executed and must be reset to active. After undoing all the events in a chunk, the next event pointer of the associated train needs to be set to the event after the first event in the chunk. The first event of the chunk is only removed from the candidate list when the entire chunk is fully allocated, so it does not have to be re-added.

Therefore, the nodes in the search tree represent chunks of events. We make scheduling decisions on the chunk granularity.

3.3 GPU algorithm

The main idea is to use the GPU to explore multiple solutions in parallel, one solution per thread. We select this granularity mainly because of the lack of parallelism in exploring a solution. To us, it is the natural and easy choice. Exploring one solution is sequential. Candidate events are chosen and scheduled until there are no more candidates. It is not possible to split this work over a number of threads since there are dependencies among the events. The kth event can not be scheduled until the (k−1)th event has been scheduled and so on.

An important difference between the GPU algorithm and the sequential algorithm is that the GPU threads does not perform any deadlock analysis. If no
track is available for scheduling, scheduling of that event is simply delayed and the next candidate event is considered instead. If none of the events in the candidate list can be scheduled, the solution branch is deadlocked and discarded. When this happens, the thread associated with that branch simply aborts by returning from the kernel function.

For a high level overview of the entire process, see figure 3.3.

3.3.1 Memory layout of the GPU implementation

One solution per thread means that little data can be shared among threads. Instead, every thread needs its own separate range of memory to work on. This leads to large memory requirements. The only data that can be shared across threads is the input, i.e. the infrastructure and original schedule. We dedicate two data structures for the read-only parts of the input data; one for event definitions and one for segment definitions.

Every event contains some data that is basically read only. This includes what segment the event is associated with; an event can never be scheduled on another segment than the one originally specified in the dataset. This is part of the problem formulation. The originally planned departure time cannot be changed either.
Chapter 3. Method

These read-only properties are separated out into a class that we call EventDef, as in "event definition":

```c
// Read only parts of an event
typedef struct {
    unsigned int startTimePlanned;
    unsigned int stopTimePlanned;
    unsigned int preferredTrack;
    unsigned int minTime;
    unsigned int trainIndex;
    unsigned int segmentIndex;
    unsigned int direction;
} EventDef;
```

Event definitions are stored as a contiguous array in global device memory. Every array element holds an instance of EventDef, and all threads share the same instances as they are read only.

It is enough to have a single instance of this class for every event, since no threads write to it. The point with this design is mainly to decrease the memory requirements, potentially allowing for increased parallelism since more models can fit into memory. But the goal was also to fit it in constant device memory, which can be heavily cached by the hardware. However, we have so far been unable to fit it into constant memory. It might be possible if the data is packed tightly together though.

Segments also contain read-only data in the form of type (station or line segment) and number of tracks. We have separated this from the dynamic allocation lists and created the class SegmentDef:

```c
// Read only parts of a segment
typedef struct {
    int type;
    unsigned int numTracks;
} ...
```

As with the event definitions, only one instance of the segment definitions are needed, regardless of the number of threads used. And we managed to fit all of the segment definitions into constant device memory, allowing for fast read access.

The segment allocation lists hold all of the events allocated to each respective segment. For the sequential CPU solution, one list per track per segment is used. This works fine, because we can dynamically increase the capacities of these lists. On the GPU however, all memory requirements are typically allocated before
the kernel is run. Consequently, the memory requirements need to be known before running the kernel. It is impossible to know how many events that will be allocated to each track. However, we do know exactly how many events that are going to be allocated to a segment. Therefore we use one allocation list per segment, regardless of the number of tracks available on that segment. This is an optimization of the memory usage. The drawback with this design is that checking the availability of a particular track is less efficient: now we need to iterate over all of the allocations instead of just the allocations for that particular track.

Since segment allocation lists are modified by the algorithm they can not be shared. Each thread uses its own set of segment allocation lists in order to find unique solutions. Every list allocates two bytes at offset zero to hold a counter for the number of elements. The entire set of all allocation lists for all threads is allocated in a single contiguous chunk of memory with a single call to cudaMalloc. Figure 3.5 illustrates how segment allocation lists are laid out in global device memory.

If we know the number of segments (|S|), the maximum number of events per segment, the number of threads per block and the base address of the memory block containing the allocation lists, the allocation list for a segment \( s \in S \) of a thread with index \( m \) in the grid can be accessed in the following way.

\[
\text{listAddress}_{s,m} = \text{base\_address} + m \times \vert S \vert \times \text{bytesPerList} + s(\text{bytesPerList})
\] (3.1)

Where \( m = \text{blockIndex} \times \text{threadsPerBlock} + \text{localThreadIndex} \) and \( \text{bytesPerList} = 2(\text{eventsPerSegment} + 1) \). In CUDA-C this can be written as:

```c
const unsigned int MAX_EVENTS_PER_SEGMENT = 56;
const unsigned int ALLOCLIST_STRIDE = MAX_EVENTS_PER_SEGMENT+1;
const unsigned int ALLOCLIST_SOL_STRIDE = NUM_SEGMENTS*ALLOCLIST_STRIDE;
uint16_t *allocList = dg_segAllocLists+
    solIdx*ALLOCLIST_SOL_STRIDE+
    evtDef->segmentIndex*ALLOCLIST_STRIDE;
```

The elements of the segment allocation lists hold 16bit unsigned integers. The algorithm uses them as indices into the list of event definitions and the lists of events, which we describe next.

The class `GpuEvent` holds the modifiable parts of an event. Each thread needs its own instance of every event. The class is defined in the following way:

```c
// Dynamic part of an event
typedef struct
{
    unsigned int startTime;
    unsigned int stopTime;
`
unsigned int track;
unsigned int state;
} GpuEvent;

Having events separated into dynamic and static parts requires some more initial setup, but both parts can be accessed using the same index. The static part of event \( k \in E^{\text{all}} \) can be accessed by adding the index to the base address: baseAddress + 28\( k \). The index is multiplied by 28 because that is the size of one EventDef instance. C compilers handle the multiplication implicitly when doing pointer arithmetic. CUDA-C works in the same way.

The candidate lists are all packed into a single array. Each thread has its own dynamic candidate list with a static capacity. The threads use their global thread id to skip forwards to their respective candidate lists. See figure 3.4.

Figure 3.4: Candidate lists. Every thread has its own candidate list. Every candidate list starts with a counter for the number of candidates currently in the list. The candidates are simply indices into the event definition list and into the event lists.

As the solution space is explored and events are being allocated to segments, nodes are being added to the current solution branch. The nodes are represented by the class GpuNode which is defined in the following way in CUDA-C:
Figure 3.5: Segment definitions and allocation lists. Every thread has its own set of segment allocation lists, one per segment. The read-only properties of segments are stored in an array shared between all threads. In this example, the allocation lists currently hold one allocated event each.

typedef struct
{
    uint16_t firstEvent;
    uint16_t lastEvent;
} GpuNode

Each node represents a sequence of events for one train. The events in this sequence have been allocated directly after each other. The nodes in the active solution branch are stored in an array. The threads use their thread IDs to skip forward to their respective solution branches. Storing the solution branches enables backtracking and also serves as representations of the solutions. Together the nodes for one complete solution represent the order with which the events were selected for allocation. Replaying the events in the order specified, using the same model/algorithm, will always yield the same train schedule.
3.3.2 Preparing the input to the GPU

Before the GPU algorithm can start running, the input needs to be prepared and transferred from host memory to device memory. This also involves allocating device memory [10].

First, the host application reads an infrastructure definition and a timetable. The timetable defines the trains and their events, see section 3.1. Then, a disturbance scenario is introduced by modifying the dataset. The disturbance scenarios that we use are described in section 3.4. After that the sequential algorithm is run partially to a preconfigured depth. We define the depth as the number of decision nodes in the branch. When this depth is reached, the sequence of events in the branch are written to an array. The array contains the *global event indices* of the events. Then, the sequential algorithm backtracks one node and explores another child node. If successful, the new, slightly different sequence of events is stored. This backtrack - explore - store pattern is repeated until a requested number of partial solutions have been generated. The process is illustrated in figure 3.6.

![Diagram](image)

Figure 3.6: Finding the starting points.

When all the partial solutions have been generated, they are transferred to the global device memory. Global device memory is allocated using *cudaMalloc* and the data is transferred using a single call to *cudaMemcpy*. See the function *uploadStartingPoints* in *ls.cu* in the source code distribution.

The segment definitions are also uploaded. First an array of *SegmentDef* is allocated in host memory. It is then populated with one element at a time by converting instances of the *Segment* class used by the sequential algorithm to *SegmentDef* instances. This conversion is performed by the wrapper. The *Segment* class holds both the constant and dynamic properties of segments.
Chapter 3. Method

Only the constant properties type and number of tracks are extracted and written to the SegmentDef instances. When the array of SegmentDef instances in host memory is fully written, it is copied to constant device memory using cudaMemcpyToSymbol. Recall that SegmentDefs can be shared among GPU threads, so only a single instance per segment is needed. After the transfer, the array of segment definitions in host memory is deallocated.

The host application then uploads the event definitions in a similar way. It uses the wrapper to look up the number of events in the dataset. The application allocates host memory to hold one EventDef instance per event. It copies the constant properties from the Event to the EventDef array in host memory. It then allocates an equally large chunk of global device memory and copies the EventDefs to it. The implementation uses global device memory since the event definitions did not fit in the constant device memory. The array in host memory is deallocated after the data transfer.

The host application writes the dynamic properties of all events to an array in host memory before uploading it to global device memory. It only uploads one set of events. Then it makes additional copies, one for every GPU thread, using multiple calls to cudaMemcpy with the cudaMemcpyDeviceToDevice flag. The application uploads the trains in a similar fashion. First it transfers one instance of every train to global device memory via a host memory buffer. Then it makes one copy per GPU thread.

We allocate global device memory so that each GPU thread gets its own candidate list. A candidate list in the CUDA implementation has one 16bit event index for every candidate and 16bits for the current number of candidates. A candidate list has at most as many candidates as there are trains in the dataset. So we allocate \((n_{\text{trains}} + 1) \times 2\) bytes per GPU thread. We then reset this memory to zero using cudaMemcpy.

Every segment must have its own allocation list. We allocate those in one chunk of memory and fill them with zeroes. The number of bytes required is calculated as size = numThreads * numSegments * (maxEventsPerSegment + 1) * 2. We add 1 to the maximum number of segments because we use an extra element in each list to hold the current number of allocations in that list. We multiply by two since we store 16bit event indices in the allocation lists (so every element is 2 bytes).

Since we need some way to store the solutions that the threads generate, we store the nodes of the current branch for each thread in an array. We need to know the size of this array since we allocate it up front. Once the CUDA kernel is running on the GPU we have no way of dynamically increasing the size. This is true for all the data structures that we use. It is possible to calculate the number of nodes required to make up a complete solution, but we do not do that. Instead we use the total number of events in the scenario to define the maximum number of solution nodes, max nodes per thread = number of events. This can be improved. Since most nodes in our scenarios include two or more events, the array
should only need space for $|T|$ elements or less. Decreasing the memory usage can lead to increased performance because we can increase the level of parallelism and achieve better latency hiding.

We allocate the array using `cudaMalloc`, fill it with zeros with `cudaMemset` and store a pointer to it in constant device memory using `cudaMemcpyToSymbol`. The nodes in this array are of type `GpuNode`, defined in the following way in CUDA-C:

```c
typedef struct {
    uint16_t firstEvent;
    uint16_t lastEvent;
} GpuNode;
```

`firstEvent` is the index of the first event in the sequence represented by the node. `lastNode` is the index of the last. In addition to the array for holding the nodes for each of the branches, we allocate an array to hold the current node counts for all of the branches. The size of this array is `number of branches`*2, i.e. one two-byte element per branch. We store a pointer to the second array too in constant device memory.

With all of this setup, allocation of device memory and data transfers from host memory to device done, we can execute the CUDA kernel on the GPU.

### 3.3.3 CUDA-C Kernel

The CUDA-C kernel is the function that every thread runs. Using the global thread index, it identifies what parts of memory it is to work on. It implements two phases. First, it uses the uploaded event indices to allocate events in the order determined sequentially before launching the kernel. It does not use any candidate list for this step. Then, it schedules the rest of the events using a candidate list and heuristics. See figure 3.3. Once the candidate list for a thread is empty or a deadlock is encountered, the thread is aborted. The high level control flow for the second step looks like this:

```c
while candidate list not empty do
    for each candidate c do
        success <- chunkAllocate(c)
        if success do
            removeCandidate(c)
            addNextCandidateForTrain(train_of(c))
            break
        done
    done
if not success do
```
Table 3.1: Disturbance scenarios.

```
Scenario # | Train id   | Segment                  | Delay (minutes) | $T_0$
---------|------------|--------------------------|----------------|--------
  1       | #93        | AK_285-AK_283 (NWK)      | 40             | 13:47:57
  2       | #9914      | Rensjön (LOC)            | 32             | 13:47:57
  3       | #9911      | Katteriåk (LOC)          | 15             | 14:50:00
  4       | #9911      | Katteriåk (LOC)          | 30             | 14:50:00
  5       | #9911      | Katteriåk (LOC)          | 45             | 14:50:00
  6       | #9948      | Gällivare (LOC)          | 15             | 10:39:00
  7       | #9948      | Gällivare (LOC)          | 30             | 10:39:00
  8       | #9948      | Gällivare (LOC)          | 45             | 10:39:00
  9       | #9912      | Kiruna malmbangård (LOC) | 15             | 11:31:00
 10      | #9912      | Kiruna malmbangård (LOC) | 30             | 11:31:00
 11      | #9912      | Kiruna malmbangård (LOC) | 45             | 11:31:00
 12      | #9913      | RGN-BJF_385 (NWK)        | 15             | 16:31:00
 13      | #9913      | RGN-BJF_385 (NWK)        | 30             | 16:31:00
 14      | #9913      | RGN-BJF_385 (NWK)        | 45             | 16:31:00
 15      | #9914      | Rensjön (LOC)            | 15             | 13:47:57
 16      | #9914      | Rensjön (LOC)            | 45             | 13:47:57
```

3.4 Disturbance scenarios

The scenarios are presented in table 3.1. All these scenarios have a time horizon of 4 hours. Given a scenario start time $T_0$, we define a scenario end time as $T_0 + 4$ hours. A scenario starting at 14:50 will have an end time of 18:50. We remove any events with a planned start time after the scenario end time as a preprocessing step. We also remove any events with a planned end time before the scenario end time. After these steps, we also exclude any trains left without events.

3.5 Experimental setup

We use the computer system described in table 3.2 for running the experiments. We create a program to run the parallel algorithm. This program accepts a number of parameters including branching depth and number of threads. Scenarios are also defined using parameters to the program. We then automate the tests using GNU Bash\(^1\). This way we can iterate over the 16 disturbance scenarios,\(^1\)

Table 3.2: Test system used in the experiments.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Debian 8</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel Core 2 Duo E6850</td>
</tr>
<tr>
<td>Memory</td>
<td>4GB</td>
</tr>
<tr>
<td>GPU</td>
<td>Nvidia GTX660 2GB</td>
</tr>
</tbody>
</table>

and collect multiple samples. For comparison, we created a very similar program for running the sequential algorithm.

For the performance measurements, we use the `gettimeofday` function to get the time before and after the code that we want to time. When measuring kernel execution time, we use the CUDA API function `cudaDeviceSynchronize` to wait for the kernel execution to stop.

We study the relationship between the independent variables `number of threads`, `threads per block`, `scenario` and `branching depth` and the dependent variables `nodes visited per time unit`, `kernel execution time`, `number of deadlocked branches` and `overhead`.

**Cost** We define a cost function as the sum of the delays for each train in seconds. This is calculated for each solution at the very end of the kernel execution.

3.6 Validity threats

The biggest internal validity threat that we identify is that our implementation is incorrect. The software developed for this thesis is implemented in 4500 lines of code. The CUDA code (which is the most difficult to read and understand) alone is written in 1800 lines of code. With all the scripts used for debugging and test automation included there are 5500 lines of code. None of this code has been reviewed or unit-tested. What we have done is extensive and careful debugging, coupled with manual review of found schedules. But it would be unrealistic to expect such a complex software to be completely without errors.

We also identify two external validity threats. Some of our experiments took around 10 hours to complete, usually running during night time. We do not know for certain that they were not interfered by background tasks running on the test system. It is possible that other processes used up system resources such as CPU time or memory during our experiments. We have disabled the screen saver and made sure there are no cron jobs scheduled.

The other external validity threat is that we rely on the `gettimeofday` function when we measure execution time. The resolution of this clock can have a

---

2http://linux.die.net/man/2/gettimeofday (2015-10-04)
big impact on the measurements. In order to counter this threat, we compute averages from multiple samples in all of our experiments.
Chapter 4

Results

We have verified manually that both the sequential and parallel algorithms find valid solutions for all scenarios. We verified the solutions by looking at filtered and sorted out, and using a GUI tool\(^1\). We also studied scenarios 1 and 2 closer for the parallel algorithm. We do not focus so much on the cost at this point as the GPU algorithm lacks a mechanism to systematically try various branching depths.

But, looking at the costs of solutions found for scenarios 1 and 2 with a branching depth of 10 we can see that the parallel algorithm finds schedules of varying cost. Tables 4.1 and 4.2 shows the 10 best solutions for scenarios 1 and 2, respectively. The *redundancy* column refers to the number of solutions found with the same cost. These tests were run with 4096 threads. All of these solutions are equal to or worse than the solution found by the sequential algorithm when branching at depth 10. But we have seen that for some branching depths the parallel algorithm finds better solutions, given enough threads. For instance, in figure 4.1 we show the results of varying the number of threads with a branching depth of 115 for scenarios 1 and 2.

4.1 Experiment 1: Number of deadlocked branches

We counted the number of deadlocked branches when branching at depth 10 using an increasing number of threads. This is an attempt to provide insight into the ramifications of discarding deadlocked branches instead of backtracking them. How many threads do we waste exploring solutions that deadlock? We show the results in figure 4.2.

4.2 Experiment 2: Performance and scalability

In order to answer research question 1 we experiment with varying the number of threads from 1024 to 22528 using seven different block configurations; 64, 96, 128, 160, 192, 224 and 256 threads per block, while also measuring the execution time

\(^1\)TrainMaster, by David Olandersson and Johanna Törnquist Krasemann
### Table 4.1: Costs of solutions found for scenario 1 when branching at branching depth 10.

<table>
<thead>
<tr>
<th>Redundancy</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1256</td>
<td>5269</td>
</tr>
<tr>
<td>1</td>
<td>6459</td>
</tr>
<tr>
<td>1770</td>
<td>7026</td>
</tr>
<tr>
<td>1</td>
<td>7300</td>
</tr>
<tr>
<td>1</td>
<td>7454</td>
</tr>
<tr>
<td>72</td>
<td>8163</td>
</tr>
<tr>
<td>2</td>
<td>9057</td>
</tr>
<tr>
<td>2</td>
<td>9324</td>
</tr>
<tr>
<td>72</td>
<td>9497</td>
</tr>
<tr>
<td>1</td>
<td>9785</td>
</tr>
</tbody>
</table>

### Table 4.2: Costs of solutions found for scenario 2 when branching at depth 10.

<table>
<thead>
<tr>
<th>Redundancy</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4413</td>
</tr>
<tr>
<td>1770</td>
<td>4980</td>
</tr>
<tr>
<td>1256</td>
<td>5948</td>
</tr>
<tr>
<td>1</td>
<td>6211</td>
</tr>
<tr>
<td>2</td>
<td>7011</td>
</tr>
<tr>
<td>121</td>
<td>7874</td>
</tr>
<tr>
<td>1</td>
<td>7979</td>
</tr>
<tr>
<td>89</td>
<td>8171</td>
</tr>
<tr>
<td>1</td>
<td>8564</td>
</tr>
<tr>
<td>72</td>
<td>8842</td>
</tr>
</tbody>
</table>
Figure 4.1: Cost of best solutions found for scenarios 1 and 2 with 115 as branching depth.
Chapter 4. Results

Figure 4.2: Number of deadlocks with increasing number of threads, branching at depth 10.
of the kernel. We want to show the relationship between the number of threads used (independent variable) and the kernel execution time (dependent variable). We only do this for scenario 1 as it is a time consuming experiment. We show the results in figure 4.3.

When running with 22528 threads, our algorithm uses 1.82GB of global memory on the graphics card.

### 4.3 Experiment 3: Data transfer overhead

Furthermore, we show the total execution time versus the kernel execution time when running with 4096 threads in total and 64 threads per block, in an attempt
4.4 Experiment 4: Performance comparison between sequential and parallel algorithm

We compare the performance of the parallel algorithm with the sequential algorithm on which it is based. Since we explore one solution per thread in the parallel algorithm and the sequential algorithm only explores one solution, we introduce the measure \textit{nodes visited per second}. The results are shown in figures 4.5, 4.6, 4.7 and 4.8. Figure 4.9 shows the results when the exploration speed for the parallel
Figure 4.5: Performance comparison between the parallel algorithm and the sequential algorithm using 22528 threads and a branching depth of 50.

The algorithm is calculated without taking the overhead of finding starting points and data transfers into account.
Figure 4.6: Performance comparison between the parallel algorithm and the sequential algorithm using 22528 threads and a branching depth of 100.
Figure 4.7: Performance comparison between the parallel algorithm and the sequential algorithm using 22528 threads and a branching depth of 150.
Figure 4.8: Performance comparison between the parallel algorithm and the sequential algorithm using 22528 threads and a branching depth of 200.
Figure 4.9: Performance comparison based on kernel performance without overhead. Branching depth 200.
Initial experiments show a number of things. The algorithm does find valid schedules on all 16 scenarios. It is also evident that there are many redundant solutions. For instance, in table 4.2, we can see that there are 1770 branches at depth 10 for scenario 2 that have a cost of 4980 (4980 seconds of total delay). So we are using 1770 threads to check practically equal solutions. The redundancy comes from the fact that the children at all nodes in the search tree represent order of scheduling or prioritization.

So, at every node, the algorithm is choosing between a number of events from different trains using the candidate list. Normally it picks the event that can be scheduled soonest first, unless scheduling fails. But at the specified branching depth, we explicitly instruct the algorithm to independently explore $n$ branches using the GPU. In other words, we explore $n$ candidates at this depth. If candidate $i$ is unrelated to candidate $i - 1$, the internal scheduling order of these events is not important. But the algorithm still treats them as unique branches. This seems to happen often given the high levels of redundancy that we found in the experiment.

The redundancy is a big problem with this algorithm. But this problem is inherent in the model, and not a drawback with heterogeneous computing. We basically model the re-scheduling problem as a search for the best order in which to schedule events. Changing that order does not always result in different schedules.

But figure 4.1 shows that at certain key branching depths, one or more of the branches lead to decreased delays, i.e. better schedules. But it is also crucial to explore enough branches. For scenario 2, we find a major improvement when using 7168 threads instead of 6144; roughly 3000 seconds of delay instead of 5000. That is a 40% improvement. The problem is to find the key branching depths.

Experiment 1 (figure 4.2) shows the number of deadlocked branches at depth 10 in the search tree. Scenarios 6, 7 and 8 stands out with a smaller increase in number of deadlocks than the others. The growth appears roughly linear for all scenarios, with some notable fluctuations. We argue that in relation to the number of branches explored, the number of deadlocks is small enough to make it viable to discard deadlocked branches.
Chapter 5. Analysis

An alternative strategy would have been to continuously backtrack and try other child nodes in order to avoid the deadlocks. Or use some form of deadlock analysis to avoid "risky" branches. The motivation for discarding deadlocked branches is to try to minimize thread divergence. But, there is a risk that these deadlock situations needs to be handled in order to find the best solutions.

Experiment 2 (figure 4.3) shows the kernel execution time as a function of the number of threads used. We explore one solution per thread, so when we increase the thread count we also increase the problem size. Increasing the thread count by a factor of 22 from 1024 to 22528 only increases the kernel runtime by a factor 6.6 from 397ms to 2634ms. That means that it is advantageous to use a high number of threads.

In figure 4.3 it is also notable that for 17408, 18432 and 19456 threads, the kernel runtime reaches high execution times above 3s that does not fit in with the linear increase otherwise observed. We do not know why this happens. Each data point represents 100 samples, and the experiment has been run multiple times just to verify. Since it is a long-running experiment, it is possible that some other process on the computer system used is disturbing the experiment by using up system resources. We have tried to turn off all other programs. We also verified that no programs where scheduled to run at any specific time.

Another thing that is worth noting is how similar execution times the seven block configurations yield. At the higher thread counts 256 threads per block seems to yield slightly better results.

The global memory usage on the device is 1.82GB when running with 22528 threads. In our algorithm, the memory is what limits how many threads we can use in total, for the entire CUDA grid. And there is plenty of opportunity to decrease the memory usage further by optimizing the data structures. Some member variables use 32 bits when they could actually use far less.

Experiment 3 shows the overhead of uploading necessary data to global memory on the GPU, with 4096 branches being evaluated in parallel. See figure 4.4. The overhead is roughly around 120ms for all of the 16 scenarios. We expect minor variations since not all scenarios have the same number of events and trains. For scenario 8 the average overhead is 120ms. The kernel executes in 360ms on average. That means that the overhead is about 25% of the total runtime for scenario 8. For scenario 14 the overhead is 110ms on average while the kernel takes 760ms to execute on average. So for scenario 14 the mean data transfer overhead is only 13% of the total execution time. For the same number of threads and the same time horizon the overhead seem to be fairly constant across scenarios.

We use cudaMemcpy for all initial setup before launching the kernel. We even use it to make copies of data that is already uploaded to global memory on the device. It could be interesting to see if the data transfer/setup overhead can be decreased by implementing some device-to-device copying in a kernel. Such a kernel could probably take advantage of shared memory since all threads would need to read the same data and then write to separate areas of global memory.
Experiment 4 compares the performance of the parallel algorithm to the performance of the sequential algorithm by how many nodes they visit per second. The comparison is performed across all 16 scenarios and over 4 different branching depths. We conduct a Mann-Whitney U test with the null hypothesis $H_0$ being that there is no difference in performance between the two algorithms. A 95% confidence level is used and the resulting p-value is $3.327 \times 10^{-9} < 0.05$. Therefore we can reject $H_0$ and draw the conclusion that the parallel algorithm visits more nodes per time unit than the sequential algorithm.

At best the parallel algorithm seem to reach speedup factors of about 7 for a branching depth of 50. This is not very impressive for a an algorithm running on the GPU. The average speedup factor across the 16 scenarios is only about 5.8 and for scenario 5 it is only about 4.9. When increasing the branching depth to 100, 150 and 200 the average speedup decreases to about 5.7, 4.6 and 4.0, respectively.

With a greater branching depth, more work is performed sequentially by the CPU, as starting points are found sequentially and uploaded to the GPU. Also, there is an overlap between the work performed on the CPU and the work performed by the kernel. The starting points that are transferred to the GPU are represented by an ordered sequence of events. The GPU then schedules those events in the same order, without any heuristics or candidate lists. So the GPU finds the best tracks and adds the events to the segment allocation lists. The host does not provide the allocation lists to the device.

Because of this overlap, which we consider overhead of our implementation, the speed factor compared to the sequential algorithm decreases as we increase the branching depth. When the branching depth increases, so does this overlapping work. In addition, as the branching depth increases, the amount of work performed sequentially (finding the starting points) also increases. This is an obvious area of improvement for future projects on this topic.

To provide more insight into the performance of the parallel algorithm and the effects of the overhead on kernel exploration speed, we conduct an additional experiment with a branching depth of 200. For this experiment, we measure the exploration speed for the kernel, without taking into account any of the overhead previously mentioned. See figure 4.9. Without the overhead, the parallel algorithm seem to reach an average speedup of about 7.4 compared to the sequential algorithm. With the overhead, the speedup is only 4.0 for this branching depth.

There are implementation steps that can be optimized to reduce the overhead, such as transferring the segment allocation lists from the host after finding the starting points, or perhaps to perform some of the work of finding the starting points on the GPU itself. However, the execution time wasted on this overhead will likely become relatively smaller once a more complete GPU approach is developed. Such an approach will likely perform more work on the GPU once all the data transfers are completed. For instance it might iteratively try branching at increasing depths.
As evident in other GPGPU literature it is not uncommon with speedup factors at 40 and above [16]. So we do not think that our algorithm is making efficient use of the GPU hardware. But at the same time, we have not applied any optimization techniques. We think that some of the reasons for the speedup factor being so small might be high thread divergence, very low ratio of calculations to global memory accesses and high register usage.

We now try to answer the research questions as defined in the introduction.

1. **How does the performance of the algorithm scale with an increased number of threads and increased problem size?**
   The kernel execution time seems to increase roughly linearly as a function of the number of threads (and solutions explored). The parallel algorithm is more effective when using many threads on a large dataset than it is using a small number of threads. Block configurations seem to have little effect on the execution time.

2. **How does the parallel algorithm perform compared with the sequential algorithm, in terms of nodes visited per time unit?**
   Our experiments indicate that the parallel algorithm performs significantly faster than the sequential algorithm. For the 16 scenarios that we tested, the average speedup factor compared to the sequential algorithms decreases from about 7 at a branching depth of 50 to about 4 for a branching depth of 200.
Chapter 6

Conclusions and Future Work

In this thesis we describe a parallel GPU approach to the problem of re-scheduling trains when disturbances occur. We base the parallel algorithm on previous work on sequential algorithms for the same problem. Based on the results that we collect during the experiments described in this thesis, we argue that GPGPU approaches show promising potential to explore more solutions per time unit than sequential algorithms running on CPUs. However, the speedup factors that we measure are far from the results reported in the GPGPU literature. We do think that our implementation can be improved in many ways in order to make better use of the GPU hardware. It would also be interesting to try our algorithm on a more powerful GPU, since the GTX660 that we use in our experiments is far from the best GPU available.

But an important question that remains unanswered is how to make use of the extra performance. Provided that we use an algorithm similar to the one we describe, how can we select interesting and unique branches to explore? Is it possible to get around the redundancy problem? Should all threads branch at the same depth, as we do in our experiments? Our algorithm is limited in that it only supports branching at a depth specified at run time. We think it would be highly relevant to experiment with a high level algorithm that iteratively or to some extent in parallel tries a number of branches at increasing depths in the tree. At each depth level it would find the best candidate event to base the next iteration on. This would not require any more data transfers between device and host. Time would also be saved by allowing each thread to continue from the best branch of the previous depth.

Also, given a branching depth, our algorithm sequentially finds one starting point for each GPU thread and then uploads them to the GPU. It should be possible to simplify this by letting the threads find their own starting points based on the branching depth and their global thread indices.

Another limitation of our algorithm is that it fails to take advantage of shared memory. Shared memory is faster than global memory and allows threads in a block to collaborate. Finding ways of using shared memory, increased memory access coalescing, lower register usage and other optimizations would also be interesting directions for further research. It would also be relevant to experiment
with GPU approaches where the CPU also performs work while the CUDA kernels are being executed.

The main contribution of this thesis is the way we represent the data structures on the GPU, and the experimental results that shows that the algorithm works. The layout of the data structures is important since decreased memory usage allows for an increased number of solutions or nodes to be stored in device memory simultaneously. We store read-only data in separate data structures which allows it to be shared among all threads. This is important as we explore one solution per thread. Less memory per solution means that more threads can be used.

It must also be noted, that this is the first massively parallel train re-scheduling algorithm.
References


