Exploring the Potential of GPU Computing in Train Rescheduling

Sai Prashanth Josyula [1], Johanna Törnquist Krasemann [2], Lars Lundberg [3]
Department of Computer Science, Blekinge Institute of Technology, Sweden
[1] E-mail: sai.prashanth.josyula@bth.se
[2] E-mail: johanna.tornquist.krasemann@bth.se
[3] E-mail: lars.lundberg@bth.se

Abstract
One of the crucial factors in achieving a high punctuality in railway traffic systems, is the ability to effectively reschedule the trains when disturbances occur. The railway traffic rescheduling problem is a complex task to solve both from a practical and a computational perspective. Problems of practically relevant sizes have typically a very large search space, making it a challenge to arrive at the best possible solution within the available computational time limit. Though competitive algorithmic approaches are a widespread topic of research, limited research has been conducted in exploring the opportunities and challenges in parallelizing them on graphics processing units (GPUs). This paper presents a *conflict detection* module for railway rescheduling, which performs its computations on a GPU. The aim of the module is to improve the speed of solution space navigation and thus the solution quality within the computational time limit. The implemented GPU-parallel algorithm proved to be more than twice as fast as the sequential algorithm. We conclude that for the problem under consideration, using a GPU for conflict detection likely gives rise to better solutions at the end of the computational time limit.

Keywords
Real-time decision support, Train rescheduling, Conflict detection, Parallel algorithms, Graphics processing units.

1 Introduction

Scheduling is a frequently employed crucial operation in several sectors, e.g., manufacturing sector, railway transport sector, etc. In railway traffic network management, the ability to efficiently schedule the trains and the network maintenance, significantly influences the punctuality of trains and Quality of Service (QoS). The importance is reflected in the goal set by the Swedish railway industry stating that by year 2020, 95% of all trains should arrive at the latest within five minutes of the initially planned arrival time (Trafikverket, 2017).

In 2017, punctuality of rail passenger services in Sweden was recorded as 90.3% (Trafikverket, 2017). The punctuality of trains is primarily affected by (1) the occurrence of disturbances, (2) the robustness of the train timetables and the associated ability to recover from delays, along with (3) the ability to effectively reschedule trains within an allowable time interval, whenever disturbances occur, so that their consequences (e.g., delays) are minimized. This paper focuses on improving the ability to effectively reschedule trains during disturbances.

Day-to-day train services in the rail sector are based on preplanned railway timetables. These timetables are planned to ensure that the services are feasible, i.e., the applicable constraints are respected. Typically, such constraints enforce safety by requiring a minimum time separation between consecutive trains passing through the same railway track. A disturbance in a railway network is an unexpected event that renders the originally planned timetable infeasible by introducing 'conflicts'. A conflict is considered to be a situation that arises when two trains require an infrastructure resource during overlapping time periods in a way such that one or more system constraints are violated.

Disturbances occur due to (1) incidents such as over-crowded platform(s) that possibly lead to unexpectedly long boarding times and minor delays, or (2) larger incidents such as power shortages, signalling system failures, train malfunctions that cause more significant delays. Train timetables are planned with appropriate time margins in order to recover from minor delays. Hence, in case of a minor disturbance, the affected train(s) may be able to recover from the effects of the disturbance provided there is sufficient buffer in the original timetable. In case of a disturbance that causes a significant delay to one or more trains, conflicts arise in the original timetable and it becomes operationally infeasible.

In order to resolve a conflict, the following rescheduling tactics are frequently employed: (1) Retiming, i.e., allocating new arrival and departures times to one or more trains, (2) local rerouting, i.e., allocating alternative tracks to one or more trains, (3) reordering, i.e., prioritizing a train over another, (4) globally rerouting the trains, or (5) partially/fully cancelling the affected train services. Detecting conflicts (i.e., checking the feasibility of the timetable) and resolving them (i.e., applying rescheduling tactics to obtain a feasible timetable) during operations, constitutes real-time railway traffic rescheduling.

During a disturbance scenario, given sufficiently large computation time, the best alternative rescheduled timetable can be chosen rather unambiguously, based on the goals of the decision-maker. However, in practice, the time interval available to reschedule the railway traffic and obtain a conflict-free rescheduled timetable at the time of a disturbance is quite narrow, e.g., 10–20 seconds (Bettinelli et al., 2017). Hence, it is a challenge to quickly explore the alternative desirable solutions and consequently reach the best alternative within the available time.

According to a recent survey (Fang et al., 2015), heuristic algorithmic approaches are most frequently employed by researchers to solve real-time railway rescheduling problems. Josyula et al. (2018) present a fast heuristic search algorithm based on iteratively detecting conflicts and resolving them using chosen rescheduling tactics. While solving the real-time railway rescheduling problem, the algorithm searches the *solution space* and produces feasible revised schedules of increasing quality with passage of time.

Though faster navigation of the solution space alone does not improve the quality of the final solution obtained by a heuristic algorithm, it very likely improves the quality of the final solution obtained within a computational time limit[1]. One way to improve the speed of solution space navigation is by designing parallel algorithms (e.g., Josyula et al. (2018)) suited for parallel hardware.

This paper presents a fast conflict detection algorithm for GPUs, which in turn results in a faster navigation of solution space. By speeding up the computation of alternative revised schedules, the most desirable schedule can be obtained by the end of the computational time

[1]assuming that the computational time limit $<$ time taken by the algorithm to obtain its final solution.

limit, thus resulting in efficient real-time railway rescheduling. The GPU-based conflict detection algorithm serves as a 'building block' for parallel train rescheduling algorithm(s).

The paper is organized as follows. The next section describes the problem at hand in more detail while overviewing the related research work. Section 3 presents a basic introduction to GPUs and explores the benefits and challenges of using them. It also presents a description of the algorithm for conflict detection (the CD algorithm) and its adaptation to GPUs (the CD-GPU algorithm). Section 4 includes the following: (i) description of the experiment used to evaluate the effects of incorporating GPUs in train conflict detection, and (ii) obtained results that comprise recorded execution times of conflict detection on central processing unit (CPU) and GPU. Section 5 analyzes and discusses the results of the experiments in order to infer valid conclusions.

2 Problem description and Related work

Optimization problems of practically relevant sizes often demand significant computational resources. Real-time railway rescheduling is one such problem that requires substantial computing capabilities to be solved to completion within an acceptable time. One of the key challenges in efficient rescheduling is to quickly explore the alternative desirable solutions in the solution space and consequently reach the best alternative within the permitted time.

Recent advances in computer hardware have made powerful chips, such as multi-core CPUs and GPUs, quite affordable and available even on commonplace computers. However, in order to employ such hardware in solving optimization problems, relevant and suitable algorithms (particularly designed and implemented for such hardware) are required. Typically, parallel algorithms are designed to employ (1) multiple processing units constituting modern CPU(s), and/or (2) GPU(s). In real-time railway (re)scheduling, the potential of parallel algorithms employing multi-core CPUs has been investigated in Mu and Dessouky (2011); Iqbal et al. (2013). More recently, Bettinelli et al. (2017); Josyula et al. (2018) report significant improvements in speed (without compromising solution quality) as a result of parallelization on CPUs.

Josyula et al. (2018) devise a train rescheduling algorithm that constructs and simultaneously navigates the branches of a search tree in parallel, as illustrated in Figure 1. The search tree is represented with conflicts as the nodes and rescheduling decisions as the edges. Each node also has a revised timetable associated with it; the root node corresponding to the original, disturbed timetable. The timetable of a subsequent child node is obtained by applying the rescheduling decision represented by its incoming edge on the parent node's timetable. The conflict represented by each node is obtained by (1) generating the node's timetable, (2) detecting the conflicts (using the CD algorithm) in the timetable, and (3) selecting the earliest of the detected conflicts. For a more detailed description of the parallel algorithm, see Josyula et al. (2018).

From Figure 1, it can be seen that conflict detection is a crucial operation that is frequently performed throughout the search tree exploration. Hence, attempts to speed up such an operation to attain faster search tree explorations, are well-justified. Initial trials to speed up conflict detection in the existing parallel algorithm by creating additional CPU threads proved unfavorable. The reason is that this resulted in the algorithm creating a large, non-optimal number of total CPU threads. However, other techniques to speed up conflict detection by employing alternatives to multi-core CPUs (e.g., GPUs) remain yet to be investigated.

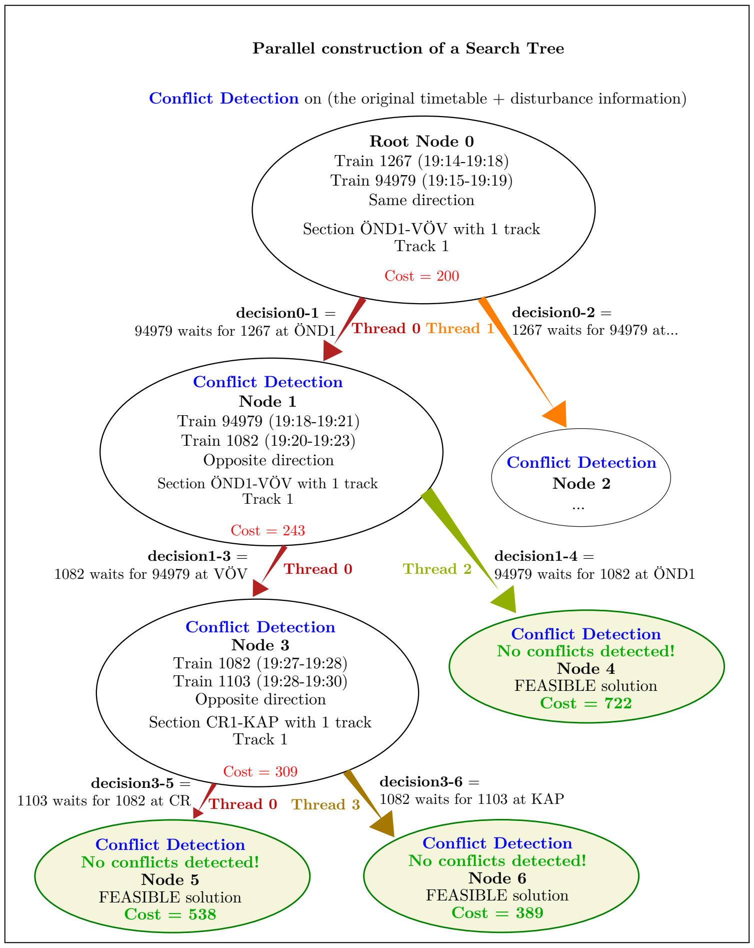


Figure 1: Illustration of the parallel algorithm designed by Josyula et al. (2018) through an example. The four parallel threads (0, 1, 2, and 3) explore the tree in parallel.

A parallel algorithm employing a GPU can either perform: (1) all of its computations on the GPU, while requiring little or no interaction with the CPU, e.g., Gmys et al. (2016), or (2) part of its computations on the GPU, while requiring significant CPU-GPU interactions. Several algorithms have been parallelized on GPUs for well-known optimization problems, such as the flow shop (Melab et al., 2012; Dabah et al., 2016), flexible job shop (Bożejko et al., 2010; Bożejko et al., 2012) and routing problems (Schulz et al., 2013). Inspired by the greedy algorithm in Törnquist Krasemann (2012), Petersson (2015) devised a building block for train rescheduling, which employs the GPU to explore multiple branches of the search tree in parallel. However, this building block spends significant time in exploring redundant solutions due to the design choices made in the search tree representation.

Very little attention has been given to employ GPUs to improve real-time railway rescheduling. Though commercial optimization solvers, such as Gurobi and CPLEX, make use of multi-core CPUs to solve a formulated model (e.g., a Mixed Integer Programming (MIP) formulation of the train rescheduling problem), currently, such solvers are not well-suited for GPUs (Glockner, 2015). For example, while solving a MIP model, each node of the search tree requires very different calculations (Glockner, 2015), whereas GPUs are designed for efficiently performing identical calculations on different data. The main objective of this research is to explore the potential of GPUs in solving the railway rescheduling problem. We did not come across research studies that answer the following research question:

How can a GPU be employed to improve computational decision support for real-time railway rescheduling?

This work contributes towards filling this research gap. The main contributions of the work presented in this paper are as follows:

(i) a building block for conflict detection on GPUs.

(ii) an evaluation of the effects of incorporating GPUs in railway rescheduling.

3 Exploring the benefits and challenges of using GPUs

A typical computer consists of a CPU as well as a GPU, both with significantly different architectures (Figure 2). A CPU is typically optimized for serial tasks, whereas a GPU is optimized for several parallel tasks. For example, consider the job of converting a color image to grayscale (Figure 3) wherein each color pixel described by a triplet of values (R, G, B) is to be converted to a corresponding grayscale pixel described by a single value that is computed by $\frac{(R+G+B)}{3}$. A GPU is highly efficient at completing this job by converting in

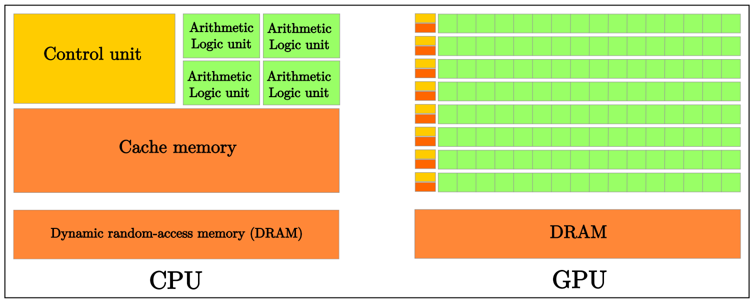


Figure 3: Conversion of a picture from color to grayscale.

context of optimization, see Brodtkorb et al. (2013). In the context of search trees, the computing power of a GPU can be utilized either for (i) parallel construction/exploration of the search tree (e.g., Petersson (2015)), or (ii) computations during tree construction/exploration (e.g., Melab et al. (2012)). The latter approach is well-motivated as the structure of the explored search tree is typically irregular, thus making tree exploration likely unfavourable for parallelization on GPUs.

In order to identify the computations worth parallelizing on a GPU, the performance reports of a previously profiled[3] heuristic algorithm for train rescheduling (Josyula et al., 2018) are examined. The results of profiling show that significant time is spent in conflict detection (the CD algorithm). While employing the algorithm to solve a rescheduling problem of moderate size[4] (i.e., a case study scenario in Josyula et al. (2018)), the conflict detection operation occurs around half a million times. Therefore, with an aim to speed up the detection of conflicts, we design a parallel algorithm for conflict detection on GPUs (the CD-GPU algorithm). Appendix A presents a code snippet[5] from the corresponding GPU program (also known as a 'kernel' in GPU terminology) implemented using the CUDA® framework (Fang et al., 2011).

Figure 4 gives an overview of the conflict detection on CPU (employing the CD algorithm) and on GPU (employing the CD-GPU algorithm) through an example. The railway infrastructure and timetable chosen for the example are illustrated in the figure. The graph adjacent to the timetable depicts that the latter is operationally infeasible and has three conflicts (labelled 1, 2, and 3). In order to detect these conflicts on a CPU, the *track event lists* are generated from the timetable, after which the CD algorithm is employed. When detecting these conflicts on GPU (by employing the CD-GPU algorithm), we instead generate *concatenated track event lists*. Then, the GPU threads, in parallel, detect the conflicts in the timetable (e.g., in Figure 4, ten threads, in parallel, detect three conflicts). In the next section, the effects of incorporating GPUs in train conflict detection are evaluated.

4 Experimental description

In order to explore the potential of GPU in solving the real-time rescheduling problem, we conduct experiments through which the speed of conflict detection on GPU is measured.

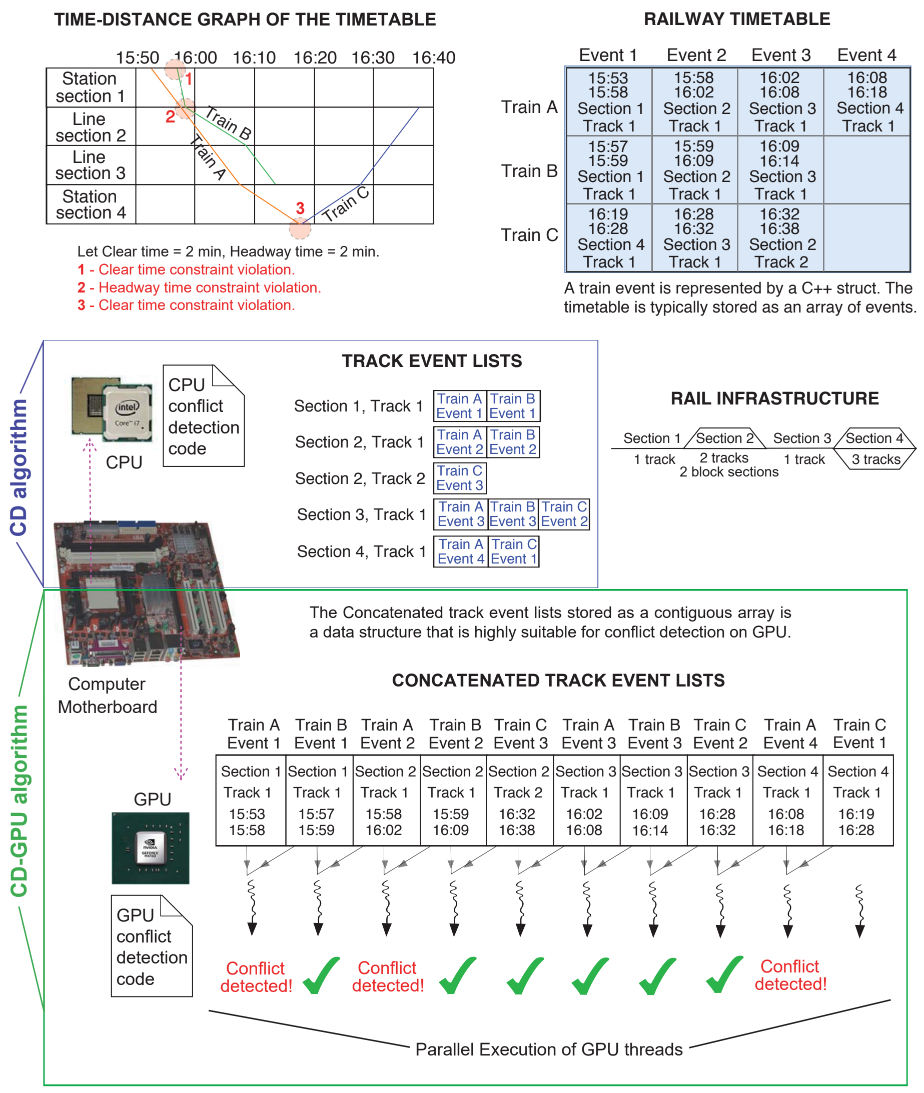
[3]using Intel® VTune™ performance profiler.
[4]59 sections, 3-hour time window, initial delay due to disturbance = 25 minutes.
[5]The entire kernel is uploaded online and is publicly available (Josyula, 2019).

Algorithm 1: The CD algorithm for conflict detection on CPU

```
Input: Timetable $T$
Output: Set of detected conflicts
1  Generate track event lists from the timetable (see Figure 4).
2  foreach section $j$ do
3      foreach track $i$ of section $j$ do
4          foreach pair of consecutive train events allocated to the track $i$ do
5              if both the trains are in the same direction and
6              the section is a multi-block section then
7                  if Headway time constraint is violated then
8                      Conflict detected between the two train events on section $j$!
9              else
10                 if Clear time constraint is violated then
11                     Conflict detected between the two train events on section $j$!
```

Algorithm 2: The CD-GPU algorithm to detect conflicts on GPU (abridged version)

```
Input: Timetable $T$
Output: Set of detected conflicts
1  Sort the timetable array to generate concatenated track event lists (see Appendix B).
2  Create $n$ threads to be executed in parallel, where $n$ = length of the array $T$.
3  i = ID of the thread, $i \in \{0, 1, 2 \ldots n-1\}$.
4  foreach thread except the last thread do
5      Event $e_i$ = $i^{th}$ element of the sorted array $T$.
6      Event $e_{i+1}$ = $i + 1^{th}$ element of the sorted array $T$.
7      if $e_i$ and $e_{i+1}$ are allocated to the same track of the same section then
8          if the trains are in the same direction and
9          the section is a multi-block line section then
10             if Headway time constraint is violated then Conflict detected!
11         else
12             if Clear time constraint is violated then Conflict detected!
```



Let Clear time = 2 min, Headway time = 2 min.
1 - Clear time constraint violation.
2 - Headway time constraint violation.
3 - Clear time constraint violation.

RAILWAY TIMETABLE

	Event 1	Event 2	Event 3	Event 4
Train A	15:53 15:58 Section 1 Track 1	15:58 16:02 Section 2 Track 1	16:02 16:08 Section 3 Track 1	16:08 16:18 Section 4 Track 1
Train B	15:57 15:59 Section 1 Track 1	15:59 16:09 Section 2 Track 1	16:09 16:14 Section 3 Track 1	
Train C	16:19 16:28 Section 4 Track 1	16:28 16:32 Section 3 Track 1	16:32 16:38 Section 2 Track 2	

A train event is represented by a C++ struct. The timetable is typically stored as an array of events.

TRACK EVENT LISTS

Section 1, Track 1	Train A Event 1	Train B Event 1	
Section 2, Track 1	Train A Event 2	Train B Event 2	
Section 2, Track 2	Train C Event 3		
Section 3, Track 1	Train A Event 3	Train B Event 3	Train C Event 2
Section 4, Track 1	Train A Event 4	Train C Event 1	

The Concatenated track event lists stored as a contiguous array is a data structure that is highly suitable for conflict detection on GPU.

CONCATENATED TRACK EVENT LISTS

Train A Event 1	Train B Event 1	Train A Event 2	Train B Event 2	Train C Event 3	Train A Event 3	Train B Event 3	Train C Event 2	Train A Event 4	Train C Event 1
Section 1 Track 1 15:53 15:58	Section 1 Track 1 15:57 15:59	Section 2 Track 1 15:58 16:02	Section 2 Track 1 15:59 16:09	Section 2 Track 2 16:32 16:38	Section 3 Track 1 16:02 16:08	Section 3 Track 1 16:09 16:14	Section 3 Track 1 16:28 16:32	Section 4 Track 1 16:08 16:18	Section 4 Track 1 16:19 16:28

Figure 4: Summary of conflict detection on CPU vs GPU.

Prior to describing the experiments in depth, it is crucial to realize the following steps that are involved in the execution of a program that employs a GPU:

1. Allocation of required resources (e.g., global memory) on the GPU.
2. Transfer[6] of input data from CPU to the allocated memory in GPU.
3. Invocation of the GPU kernel that works on the input data and outputs results.
4. Transfer of results from the memory in GPU to the CPU.

4.1 Input data

Given an initial timetable T_{init} subject to a disturbance, the algorithm outlined in Josyula et al. (2018) generates, in parallel, alternative rescheduling solutions which are computed by iterating between conflict detection and conflict resolution (i.e., rescheduling of trains). We denote an intermediary rescheduling solution that is subject to conflict detection, T. Hence, the algorithm computes in parallel a set $\boldsymbol{T}$ of alternative rescheduling solutions. For instance, in the example run of the parallel algorithm (Josyula et al., 2018) shown in Figure 1, four rescheduling solutions are being generated in parallel (i.e., four branches of the tree are being explored in parallel). Therefore, corresponding to this example, the set $\boldsymbol{T}$ consists of four timetables. In other words, $|\boldsymbol{T}| = 4$.

The purpose of the experiments is to apply the GPU-based conflict detection on the set of alternative rescheduling solutions denoted $\boldsymbol{T}$. This is accomplished through the following three steps:

(i) transferring the set $\boldsymbol{T}$ from the CPU to the GPU,

(ii) detecting in parallel, conflicts in each timetable T, on the GPU,

(iii) transferring the results from GPU to CPU.

The potential of GPU can be best measured when the above steps (i)–(iii) are carried out a considerable number of times (e.g., 5000 times). This is taken into consideration while recording the execution times.

The size of results transferred in step (iii) is proportional to the size of the input data transferred in step (i); it is not related to the number of conflicts detected by the CD-GPU algorithm. The reason is that the results comprise values that correspond to each train event of the input data. These values indicate the presence/absence of a conflict along with its type (conflict due to violation of headway time constraint or clear time constraint). Similarly, the time taken for step (ii) (the CD-GPU algorithm) depends on the number of train events, not the number of conflicts in the input timetable(s). For instance, the CD-GPU algorithm requires equal execution time in the following two cases:

- to determine that an input feasible timetable has zero conflicts,
- to determine the number of conflicts in an input infeasible timetable.

[6]Typically, CPU communicates with GPU via high-speed bus called PCI express.

Due to the above reasons, the input data used throughout the experiments is generated in the following way. A feasible timetable T_{init} consisting of 740 train events is randomly chosen. When subject to a random disturbance of five minutes, 13 conflicts arise in T_{init}. This disturbed timetable consisting of 13 conflicts is used for populating the set $\boldsymbol{T}$ throughout the experiments. The railway infrastructure consists of 59 sections (including stations) and extends from Karlskrona to Tjörnarp.

4.2 Experimental variables

Variable	Description	Type (Independent, Controlled or Dependent)
$\|\boldsymbol{T}\|$	Number of timetables in the set $\boldsymbol{T}$.	This is an independent variable, the value of which is systematically changed.
t	Total number of times steps (i)–(iii) are executed. The value of t = 10,000.	The value of this variable is intentionally kept constant in order to clearly isolate the relationship between the other variables. This is the controlled variable.
c	Total number of times the conflict detection is performed $(\|\boldsymbol{T}\| \times t)$.	This value is systematically changed to see its effect on the recorded measurements. This is the independent variable.
t_{gpu}	Time taken by GPU to perform conflict detection c times.	The value of this variable is observed and recorded. This is the dependent variable.

Table 1: Variables used in the experiments.

Table 1 lists the experimental variables and describes them in detail. As a benchmark for the recorded values of t_{gpu}, the associated conflict detection computations on the CPU are performed by:

(I) detecting conflicts in the chosen timetable T,

(II) recording the execution time (t_{cpu}) taken by the CPU to perform step I c times.

$$\text{Speedup } (S) = \frac{\text{Time taken by CPU to perform conflict detection } c \text{ times}}{\text{Time taken by GPU to perform conflict detection } c \text{ times}} = \frac{t_{cpu}}{t_{gpu}}$$

Note that each value of $|\boldsymbol{T}|$ in the performed experiments is intended to represent the number of branches of the search tree that a train rescheduling algorithm explores in parallel. Hence, the values are limited to $|\boldsymbol{T}| = \{1, 2, 4, 8, \ldots, 256\}$[7]; for practical problem scenarios, it is quite realistic to explore up to 256 branches of the search tree in parallel. The measurements for $|\boldsymbol{T}| = 512$ are recorded only to notice the trend of speedup.

4.3 Platform description

The experiments are performed on a laptop equipped with an Intel Core i7-8550U CPU and an Nvidia® GPU with compute capability 6.1. The GPU consists of 3 streaming multiprocessors (SMs), each with 128 cores. For detailed specifications of the GPU, see Appendix C.

[7]For the sake of convenience, we use only powers of 2.

The underlying operating system is 64-bit Windows® 10 Education and the available random-access memory is 16 GB[8]. The CPU code has been compiled using Microsoft® C++ optimizing compiler V19.14.26431, with whole program optimization (`/GL` flag) and maximum optimization favouring speed (`/O2` flag). The GPU code has been compiled using Nvidia CUDA compiler V9.2.148.

4.4 Kernel launch parameters

In an Nvidia GPU, the basic unit of execution is a *warp*, which is a collection of several threads. For devices with compute capability 6.1, a warp consists of 32 threads. All the threads in a warp are executed simultaneously by an SM; multiple warps can be executed on an SM at once.

A *block* of threads is a CUDA programming abstraction; all the threads in a block can communicate with each other (via shared memory, synchronization primitives, etc.) to co-operatively solve a problem in parallel.

In order to execute the conflict detection kernel on GPU, the *number of threads per block* and the *total number of blocks* need to be specified. These are known as kernel launch parameters. A frequently employed heuristic to select the number of threads per block is to aim for a high *occupancy*.

$$\text{Occupancy} = \frac{\text{number of warps running concurrently on an SM}}{\text{maximum number of warps that can run concurrently on the SM}} \tag{1}$$

The CUDA occupancy calculator (Nvidia, 2019) allows computation of the occupancy of a GPU by a given CUDA kernel.

For the GPU used in the experiments, the denominator of Equation 1 is 64. Compiling the conflict detection kernel with the compilation flag `--ptxas-options=-v` shows that it uses 25 registers per thread and 18960 bytes of shared memory per block. When this kernel resource usage is given as input to the occupancy calculator, Figure 5 is obtained as output. Based on this figure, the number of threads per block is chosen to be 512 in order to achieve 100% occupancy. The number of blocks to be launched is calculated using the following formula:

$$\text{Number of blocks } (b) = \frac{\text{Total number of threads}}{\text{Number of threads per block}} = \frac{\text{Total number of threads}}{512}$$

From Algorithm 2 and Figure 4, notice that the total number of GPU threads is equal to the total number of events involved in conflict detection. In the experiments, the latter number is supposed to be the number of events in set $\boldsymbol{T}$, which is $|\boldsymbol{T}| \times 740$. However, since $|\boldsymbol{T}| \times 740$ is not always an integral multiple of 512, the number of blocks are determined using the following formula:

$$\text{Number of blocks } (b) = \left\lfloor \frac{\text{Total number of events in set } \boldsymbol{T}}{512} \right\rfloor \tag{2}$$

Consequently, throughout the experiments, conflict detection on the GPU is not performed on all the events in the set $\boldsymbol{T}$. The last x events, where $x = (|\boldsymbol{T}| \times 740) \ \% \ 512$, are not sent as input to the GPU, and hence are not involved in conflict detection. The same events are excluded while performing conflict detection on the CPU.

[8] 1 kilobyte (KB) = 2^{10} bytes, 1 megabyte (MB) = 2^{10} KB, 1 gigabyte (GB) = 2^{10} MB.

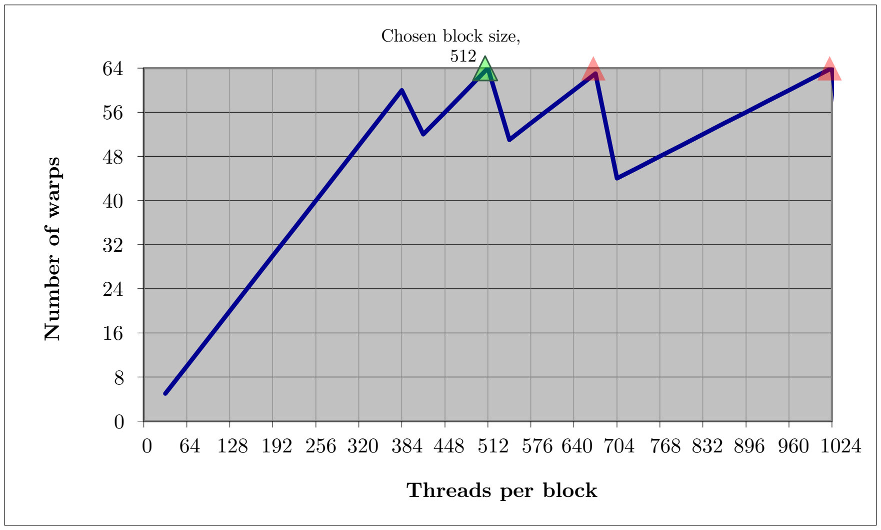


Figure 5: Impact of varying block sizes on multiprocessor occupancy.

4.5 Recorded results

The results of the experiments, summarized in Table 2, show that employing the GPU for conflict detection during real-time railway rescheduling can make the process more than twice as fast. Each recorded value of t_{gpu} and t_{cpu} is the average of five observations.

Explanation of the decrease in speedup value in Table 2: The total data[9] d transferred between CPU and GPU is proportional to $|\boldsymbol{T}|$. Through profiling the kernel, it was observed that the data transfer speed d_{speed} is not constant across different values of $|\boldsymbol{T}|$; for smaller values of $|\boldsymbol{T}|$ (consequently, smaller values of d), the d_{speed} is greater.

For example, for $|\boldsymbol{T}| = 1$, $d = 123$ MB and $d_{speed} = 6.3$ GB/sec. For $|\boldsymbol{T}| = 2$, $d = 246$ MB and $d_{speed} = 5.7$ GB/sec. For $|\boldsymbol{T}| = 256, 512$, $d = 45$ GB and 90 GB, whereas the data transfer speeds are 3 GB/sec and 2.6 GB/sec respectively. This explains the fall in speedup (from 2.77 to 2.43) when the value of $|\boldsymbol{T}|$ is increased from 256 to 512.

5 Discussions and Conclusion

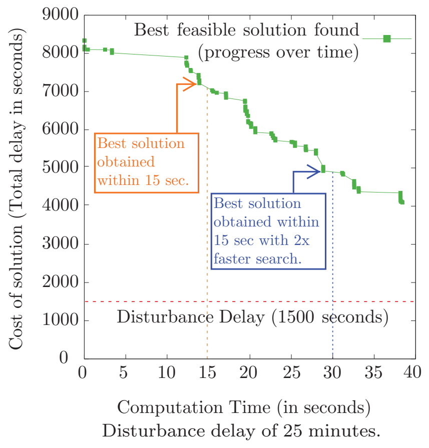
We present two examples (Figure 6) to illustrate the potential improvement (or the lack thereof) in the quality of solution due to faster search tree navigation. As can be seen in Figure 6a, a twofold faster search tree navigation leads us to obtain better solutions within a given computational time limit of, e.g., 15 seconds. However, in the disturbance scenario in Figure 6b, a twofold faster search tree navigation does not lead to a better solution within a time limit of 15 seconds.

GPUs possess the potential to speedup real-time railway rescheduling, thus improving

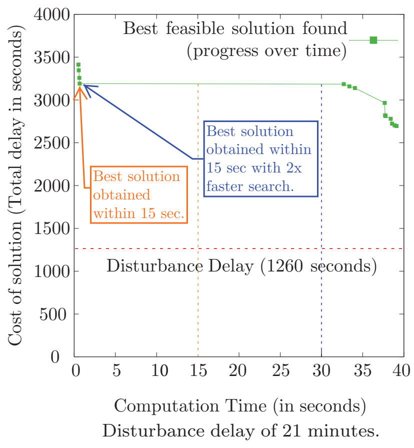
[9]Size of total data transferred = (Size of input data + Size of results) $\times 10^3$

Number of			Number of		Time (sec)		
Time-tables in set $\boldsymbol{T}$	Times conflict detection (c)	Events in set $\boldsymbol{T}$	Events used for conflict detection	Blocks (b)	t_{gpu}	t_{cpu}	Speedup
1	1×10^3	1×740	1×2^9	1	1.23	0.22	0.18
2	2×10^3	2×740	2×2^9	2	1.45	0.42	0.29
4	4×10^3	4×740	5×2^9	5	1.47	0.88	0.60
8	8×10^3	8×740	11×2^9	11	1.66	1.87	1.13
16	16×10^3	16×740	23×2^9	23	2.49	3.14	1.26
32	32×10^3	32×740	46×2^9	46	3.50	7.33	2.10
64	64×10^3	64×740	92×2^9	92	6.02	15.06	2.50
128	128×10^3	128×740	185×2^9	185	10.81	29.17	2.70
256	256×10^3	256×740	370×2^9	370	19.03	52.75	2.77
512	512×10^3	512×740	740×2^9	740	40.73	99.20	2.43

Table 2: Results of conflict detection on CPU and GPU. For each measurement of t_{gpu}, steps (i)–(iii) are carried out 10^3 times. The number of events per timetable = 740, and the number of threads per block = 2^9.

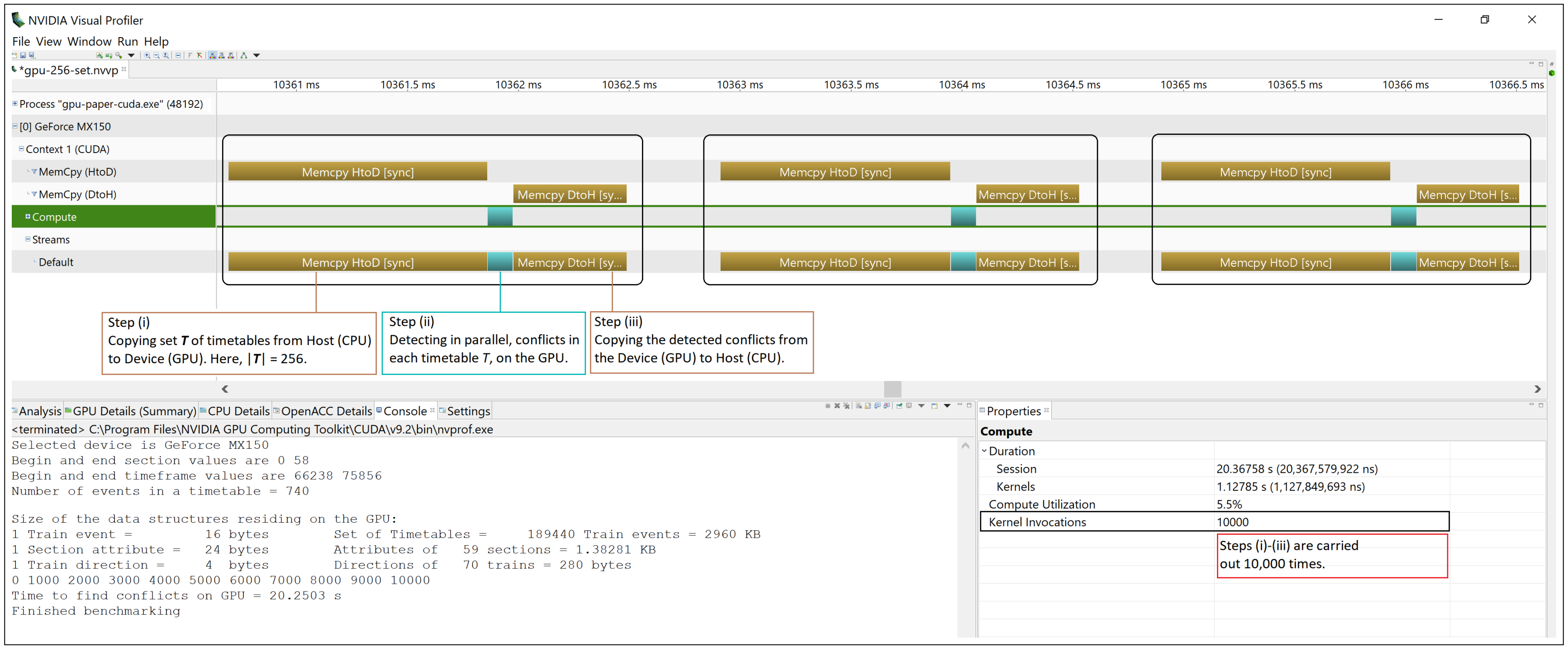


(a) Example 1: Solutions obtained by a real-time train rescheduling algorithm during a disturbance scenario.



(b) Example 2: Solutions obtained by a real-time train rescheduling algorithm during another disturbance scenario.

Figure 6: Examples to illustrate potential improvement in quality of obtained 'best' solution due to faster search tree navigation.



Step (i)
Copying set *T* of timetables from Host (CPU) to Device (GPU). Here, |*T*| = 256.

Step (ii)
Detecting in parallel, conflicts in each timetable *T*, on the GPU.

Step (iii)
Copying the detected conflicts from the Device (GPU) to Host (CPU).

```
<terminated> C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v9.2\bin\nvprof.exe
Selected device is GeForce MX150
Begin and end section values are 0 58
Begin and end timeframe values are 66238 75856
Number of events in a timetable = 740

Size of the data structures residing on the GPU:
1 Train event =          16 bytes        Set of Timetables =     189440 Train events = 2960 KB
1 Section attribute =    24 bytes        Attributes of   59 sections = 1.38281 KB
1 Train direction =      4  bytes        Directions of   70 trains = 280 bytes
0 1000 2000 3000 4000 5000 6000 7000 8000 9000 10000
Time to find conflicts on GPU = 20.2503 s
Finished benchmarking
```

Compute	
Duration	
Session	20.36758 s (20,367,579,922 ns)
Kernels	1.12785 s (1,127,849,693 ns)
Compute Utilization	5.5%
Kernel Invocations	10000

Steps (i)-(iii) are carried out 10,000 times.

Figure 7: Output from Nvidia Visual Profiler when number of timetables in the set T = 256.

the likelihood of arriving at a better solution within the computational time limit. However, results of the experiments (tabulated in Table 2) show that this potential speedup (resulting from faster conflict detection using GPUs) requires several rescheduled timetables (i.e., $\geq$ 8) to be sent to the GPU in one transfer.

Profiling[10] the parallel program with the Nvidia Visual Profiler® shows (Figure 7) that for $\boldsymbol{T}$ = 256, only 5.5% of the recorded time (indicated by the parameter t_{gpu} in Table 2) is actually spent detecting conflicts. A major portion of the recorded time is spent on transferring data between the CPU and GPU, which is a demanding side-effect of using a GPU in frequent interaction with a CPU. Since Table 2 shows that the speedup of using the GPU (including communication time) for $\boldsymbol{T}$ = 256 is 2.77, the speed up attained in conflict detection on the GPU (excluding communication time) is $\approx \frac{2.77}{0.055}$, which is ≈ 50. Hence, conflict detection on GPUs is far more efficient than reflected by the speedup values in Table 2. This indicates that massive speedups could be achieved through solution approaches that execute the entire train rescheduling algorithm on a GPU (in contrast to the presented approach of executing only the conflict detection on the GPU). Such approaches would drastically reduce the CPU-GPU memory transfers which are significant bottlenecks in the presented approach.

Thus, we conclude that it is worthwhile to investigate modifications to existing real-time railway rescheduling algorithms (e.g., Josyula et al. (2018)) such that (i) several timetables are sent to a GPU for parallel conflict detection, or (ii) the algorithm is executed entirely on a GPU.

6 Acknowledgements

The research presented in this paper has been conducted within the research project TRANSFORM, which is funded by grants from the municipality of Karlshamn as well as the Swedish Research Council (FORMAS) via ERA-NET. The project has also received support from Trafikverket (The Swedish Transport Administration) and Blekingetrafiken.

The authors would like to thank Dr. Andrew Moss, Dr. Martin Boldt and Dr. Veronica Sundstedt for providing feedback that greatly improved the manuscript. Thanks to Dr. Lawrence Henesey for providing valuable feedback on the extended abstract of this manuscript.

References

Bettinelli, A., Santini, A., and Vigo, D. (2017). A real-time conflict solution algorithm for the train rescheduling problem. *Transportation Research Part B: Methodological*, 106:237 – 265.

Bożejko, W., Uchroński, M., and Wodecki, M. (2010). Parallel hybrid metaheuristics for the flexible job shop problem. *Computers & Industrial Engineering*, 59(2):323–333.

Bożejko, W., Hejducki, Z., Uchroński, M., and Wodecki, M. (2012). Solving the flexible job shop problem on multi-gpu. *Procedia Computer Science*, 9:2020 – 2023. Proceedings of the International Conference on Computational Science, ICCS 2012.

[10]Nvidia Visual Profiler is a cross-platform performance profiling tool which provides vital feedback to developers for optimizing CUDA C/C++ programs.

Brodtkorb, A. R., Hagen, T. R., Schulz, C., and Hasle, G. (2013). Gpu computing in discrete optimization. part i: Introduction to the gpu. *EURO Journal on Transportation and Logistics*, 2(1):129–157.

Dabah, A., Bendjoudi, A., El-Baz, D., and Aitzai, A. (2016). GPU-Based Two Level Parallel B B for the Blocking Job Shop Scheduling Problem. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 747–755.

Fang, J., Varbanescu, A. L., and Sips, H. (2011). A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference*, pages 216–225. IEEE.

Fang, W., Yang, S., and Yao, X. (2015). A survey on problem models and solution approaches to rescheduling in railway networks. *IEEE Transactions on Intelligent Transportation Systems*, 16(6):2997–3016.

Glockner, G. (2015). Parallel and distributed optimization with gurobi optimizer. `http://www.gurobi.com/resources/seminars-and-videos/parallel-and-distributed-optimization`. [Last accessed 24-September-2018].

Gmys, J., Mezmaz, M., Melab, N., and Tuyttens, D. (2016). A gpu-based branch-and-bound algorithm using integer–vector–matrix data structure. *Parallel Computing*, 59(Supplement C):119 – 139. Theory and Practice of Irregular Applications.

Iqbal, S. M. Z., Grahn, H., and Törnquist Krasemann, J. (2013). Multi-strategy based train re-scheduling during railway traffic disturbances. In *Proceedings of the 5th International Seminar on Rail Operations Modeling and Analysis (RailCopenhagen 2013, pp. 387-405), Technical University of Denmark, Denmark.*

Josyula, S. P. (2019). Cuda code for conflict detection on gpus. `https://github.com/sai-j/gpu-conflict-detection`.

Josyula, S. P., Törnquist Krasemann, J., and Lundberg, L. (2018). A parallel algorithm for train rescheduling. *Transportation Research Part C: Emerging Technologies*, 95:545–569.

Melab, N., Chakroun, I., Mezmaz, M., and Tuyttens, D. (2012). A gpu-accelerated branch-and-bound algorithm for the flow-shop scheduling problem. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 10–17. IEEE.

Mu, S. and Dessouky, M. (2011). Scheduling freight trains traveling on complex networks. *Transportation Research Part B: Methodological*, 45(7):1103–1123.

Nvidia (2019). Cuda occupancy calculator. `https://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`.

Petersson, A. (2015). Train re-scheduling : A massively parallel approach using cuda. Master's thesis, Blekinge Institute of Technology, Department of Computer Science and Engineering.

Schulz, C., Hasle, G., Brodtkorb, A. R., and Hagen, T. R. (2013). Gpu computing in discrete optimization. part ii: Survey focused on routing problems. *EURO Journal on Transportation and Logistics*, 2(1):159–186.

Törnquist Krasemann, J. (2012). Design of an effective algorithm for fast response to the re-scheduling of railway traffic during disturbances. *Transportation Research Part C: Emerging Technologies*, 20(1):62–78.

Trafikverket (2017). The Swedish Transport Administration Annual Report. https://trafikverket.ineko.se/se/the-swedish-transport-administration-annual-report-2017. [Last accessed 24-September-2018].

A Fragment of code from the GPU program

Figure A: Code snippet from the GPU kernel

```
// 1D grid and 1D blocks
auto threadsPerBlock = blockDim.x;
// Local thread number in the block
auto l = threadIdx.x;
auto blockNumInGrid = blockIdx.x;
// Global thread number
auto i = blockNumInGrid * threadsPerBlock + l;

// Shared memory data structures for speed
__shared__ tr_event sh_concat_tracklists[1025];
__shared__ int sh_directions[128];
__shared__ sec_attribs sh_section_attr[128];

// Private variable (per GPU thread) to record the conflict
 ↪  event
int2 conflict;
conflict.x = -1;
conflict.y = -1;

// Copy the section attributes to the block's shared memory
if (l < numb_sections)
    sh_section_attr[l] = section_attr[l];

// Copy the train directions to the thread block's shared
 ↪  memory
if (l < numb_trains)
    sh_directions[l] = directions[l];

// Copy a 'block' of sorted section lists to shared memory
sh_concat_tracklists[l] = concat_tracklists[i];
// For the last thread in the block
if(l == threadsPerBlock - 1)
    sh_concat_tracklists[l+1] = concat_tracklists[i+1];

// Ensure all writes to shared memory are completed
__syncthreads();

// Other code not included in this snippet

// Coalesced copy the detected conflict to global memory
conflicts[i] = conflict;
```

B Efficient generation of concatenated track event lists

Concatenated track event lists (for use by the GPU) can be efficiently generated from a timetable by sorting it using the following logic.

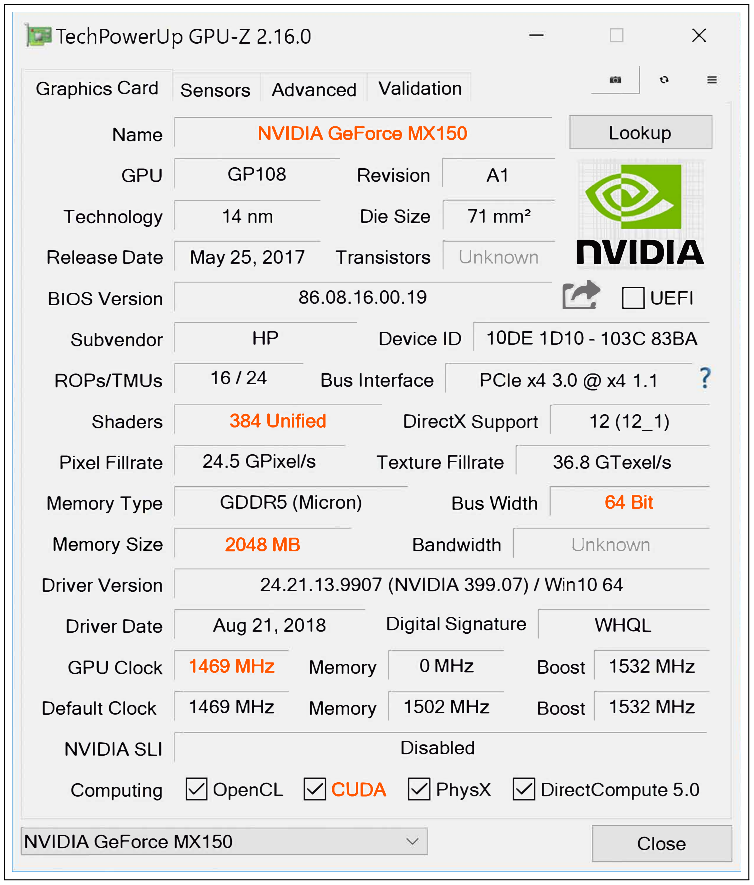
Figure A: Sorting logic for the train events comprising a timetable

```
sort (event1, event2)
{
  if(event1.section == event2.section)
  {
    if(event1.track == event2.track)
      // Sort based on begin times.
    else
      // Sort based on track numbers.
  }
  else
    // Sort based on section numbers.
}
```

C Detailed specifications of the GPU used in the experiments.

Property	Value
Number of streaming multiprocessors	3
CUDA cores per multiprocessor, total cores	128, 384
Number of threads per warp	32
Maximum warps per multiprocessor	64
Maximum blocks per multiprocessor	32
Maximum threads per multiprocessor	2048
Maximum threads per block	1024
Register size, registers per multiprocessor	32 bit, 65536
Maximum registers per block	32 bit, 65536
Maximum registers per thread	255
Register allocation unit size	256
Register allocation granularity	warp
Shared memory allocation unit size	256
Warp allocation granularity	4
Maximum shared memory per block	48 KB
Shared memory per multiprocessor	96 KB
Constant memory	64 KB
Global memory	2048 MB

Table 3: Physical limits of the GPU used in the experiments.



TechPowerUp GPU-Z 2.16.0

Name	NVIDIA GeForce MX150				
GPU	GP108	Revision	A1		
Technology	14 nm	Die Size	71 mm²		
Release Date	May 25, 2017	Transistors	Unknown		
BIOS Version	86.08.16.00.19			UEFI	
Subvendor	HP	Device ID	10DE 1D10 - 103C 83BA		
ROPs/TMUs	16 / 24	Bus Interface	PCIe x4 3.0 @ x4 1.1		
Shaders	384 Unified	DirectX Support	12 (12_1)		
Pixel Fillrate	24.5 GPixel/s	Texture Fillrate	36.8 GTexel/s		
Memory Type	GDDR5 (Micron)	Bus Width	64 Bit		
Memory Size	2048 MB	Bandwidth	Unknown		
Driver Version	24.21.13.9907 (NVIDIA 399.07) / Win10 64				
Driver Date	Aug 21, 2018	Digital Signature	WHQL		
GPU Clock	1469 MHz	Memory	0 MHz	Boost	1532 MHz
Default Clock	1469 MHz	Memory	1502 MHz	Boost	1532 MHz
NVIDIA SLI	Disabled				
Computing	✓ OpenCL	✓ CUDA	✓ PhysX	✓ DirectCompute 5.0	

NVIDIA GeForce MX150

Figure 8: Further detailed specifications of the GPU.