GPU-accelerated Model Checking of Periodic Self-Suspending Real-Time Tasks

Per-Erik Måhl and Tim Liberg
Mälardalen University
Västerås, Sweden
{pml09001,tlg09003}@student.mdh.se

Supervisor: Mikael Åsberg
Examiner: Thomas Nolte

May 31, 2012
# Contents

1 Introduction 6

2 Preliminaries 7
   2.1 Timed automata . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
   2.2 Self-suspending task model . . . . . . . . . . . . . . . . . . . . . . 8
   2.3 CUDA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
      2.3.1 Programming model . . . . . . . . . . . . . . . . . . . . . . . . . 8
      2.3.2 Memory hierarchy . . . . . . . . . . . . . . . . . . . . . . . . . . 9

3 Background 11

4 Related work 13

5 Problem formulation 14

6 Analysis of problem 15
   6.1 Choosing a language . . . . . . . . . . . . . . . . . . . . . . . . . . 15
   6.2 Parallelization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
      6.2.1 Branches . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
      6.2.2 Stopping the search . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
   6.3 Memory restrictions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
      6.3.1 Choosing paths . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

7 Method 17
   7.1 Parallelization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
   7.2 Seed system . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
      7.2.1 Seeded Tree Search (STS) . . . . . . . . . . . . . . . . . . . . . . . . . 19
   7.3 Search model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
      7.3.1 CUDA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
      7.3.2 Restrictions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
      7.3.3 Device code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
      7.3.4 Host code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

8 Solution 23
   8.1 Modified STS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
   8.2 Example of converting guard evaluations from C to CUDA C . 25

9 Results 29
   9.1 Experimental setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
   9.2 Finding feasible scheduling paths . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
Abstract

Efficient model checking is important in order to make this type of software verification useful for systems that are complex in their structure. If a system is too large or complex then model checking does not simply scale, i.e., it could take too much time to verify the system. This is one strong argument for focusing on making model checking faster. Another interesting aim is to make model checking so fast that it can be used for predicting scheduling decisions for real-time schedulers at runtime. This of course requires the model checking to complete within an order of milliseconds or even microseconds. The aim is set very high but the results of this thesis will at least give a hint on whether this seems possible or not. The magic card for (maybe) making this possible is called Graphics Processing Unit (GPU). This thesis will investigate if and how a model checking algorithm can be ported and executed on a GPU. Modern GPU architectures offer a high degree of processing power since they are equipped with up to 1000 (Nvidia GTX 590) or 3000 (Nvidia Tesla K10) processor cores. The drawback is that they offer poor thread-communication possibilities and memory caches compared to CPU. This makes it very difficult to port CPU programs to GPUs.

The example model (system) used in this thesis represents a real-time task scheduler that can schedule up to three periodic self-suspending tasks. The aim is to verify, i.e., find a feasible schedule for these tasks, and do it as fast as possible with the help of the GPU.
Preface

We would like to thank our supervisor Mikael Åsberg for great tutoring, supervision and feedback during this thesis. Regards also go out to André Hansson for letting us conduct experiments on his GPU. Last but not least, we wish to express our gratitude to the lab manager Daniel Flemström at the Ericsson Research Laboratory for letting us use their equipment.
1 Introduction

The Graphics Processing Unit (GPU) is becoming more and more focused on applications other than just graphics. Hence, we call this trend general purpose computing on GPU, i.e., GPGPU. The GPU is currently holding almost twice as much peak processing power compared to CPU and three out of the five top supercomputers in the world today are powered by GPUs [8]. There is no doubt that many-core architectures such as GPUs are advancing rapidly in the market, hence, research efforts in various areas must keep pace with the evolution of the GPU in order to make as much as possible out of today's technology and stay competitive.

The GPU offers enormous computing capacity (if used in the right way) while we also see computing problems in research areas such as software verification. One such verification technique is called Model Checking [5, 7, 23] and it has one serious drawback: it cannot verify systems in a timely manner if they are too large. It can take several years at worst. We want to investigate if the GPU can offer a remedy for search issues found in model checking.

The example problem that we will use in this thesis, in order to show the effectiveness of running model checking on a GPU, is finding schedules in a real-time task model called self-suspending tasks [9, 15, 17]. There has been plenty of work on different aspects of self-suspending tasks in the past [1, 10, 12, 13, 14, 19, 20, 21, 22]. However, to the best of our knowledge, no existing work combine self-suspending tasks with model checking and GPU programming.

To this day, there is no optimal algorithm for scheduling self-suspending periodic real-time tasks. The search problem of finding schedulable systems in the self-suspending task model has been proven to be NP-hard [22]. Feasible and sustainable schedules to such systems can be found by using timed-automata-assisted scheduling (TAAS) [1], but only off-line. A note to the reader is that the timed-automata presented in [1] cannot actually find feasible schedules. We will elaborate more on this in Section 3. In this thesis, we investigate the possibility of finding feasible schedules on such systems using a GPU. Positive results from this thesis can have great impact on scheduling in real-time systems, as well as speeding up model checking in general.
2 Preliminaries

This section will give preliminary background on timed automata and the self-suspending task model.

2.1 Timed automata

Timed automata [2] is a modelling language for modelling and analyzing software and hardware systems. Timed automata has later been extended with tasks. The task parameters included are periods, priorities, execution times etc. This model is referred to as task automata [6] and it associates tasks to the locations in a timed automata system. The task-automata model is supported by the UPPAAL-TIMES [3] tool. This tool can verify timed automata models. The example self-suspending model used throughout in this thesis was developed using UPPAAL-TIMES. The main benefit of using this tool is that it supports task automata which is typically well suited when modeling schedulers and tasks. UPPAAL-TIMES has a code-generator which has been used in this thesis to generate C-code from the self-suspending task model. This C-code has later been mapped to GPU (CUDA) source code.

Figure 1 illustrates an example of a task automaton. The left sub-figure will stay in location Location\(_1\) up until time-unit 10 (according to the guard \texttt{time==10} and the invariant \texttt{time<=10}). The transition to location Location\(_2\) will force (\texttt{channel!}) the automaton in the right sub-figure to also move its control to Location\(_2\). A task (\texttt{task1}) is released for execution when the control reaches Location\(_2\) (right sub-figure). The left sub-figure can at any time move back to Location\(_1\) again. The right sub-figure can loop the transition (\texttt{a:=a+1}) any number of times without the clock progressing since the state is marked with \texttt{U}, i.e., urgent.

![Timed automata example](image_url)  

Figure 1: Timed automata example.
2.2 Self-suspending task model

The self-suspending task model [9, 15, 17] is an extension of the well known periodic task model [11]. Periodic tasks $\tau_i$ execute periodically every period $T_i$, for $C_i$ time units based on the task priority $P_{r_i}$. A task $\tau_i$ must finish its execution (of maximum time length $C_i$) before its deadline $D_i$ within every period $T_i$ (where $C_i < D_i \leq T_i$).

Self suspending tasks are characterized with the tuple $\tau_i = \{P_i, T_i, D_i\}$ where $P_i$ is an execution pattern $P_i = \{C_1^i, E_1^i, C_2^i, E_2^i, \ldots, C_m^i\}$. Each $C_j^i$ represents an execution time and each $E_j^i$ corresponds to a suspension time. Uncertain tasks have interval values for both $C_j^i$ and $E_j^i$, i.e., execution and suspension times can vary within an interval. Hence, uncertain tasks have the execution pattern $P_i = \{[C_1^l_i, C_1^u_i], [E_1^l_i, E_1^u_i], [C_2^l_i, C_2^u_i], \ldots, [C_m^l_i, C_m^u_i]\}$.

The notations for self-suspending tasks are inspired from [1].

2.3 CUDA

CUDA, short for Compute Unified Device Architecture, is a general-purpose parallel computing architecture for GPUs developed by Nvidia. It is the architecture used in this thesis and this section will explain the basics of CUDA.

2.3.1 Programming model

A CUDA-enabled GPU chip consists of a number of multiprocessors. These multiprocessors each have a set of stream processors. These processors each run a thread and the threads are started by calling a GPU kernel-function. In CUDA, a group of threads form a thread block. All blocks together form a grid. The advantage with this model is that threads can communicate with each other using low-latency memory. Stream processors of a multiprocessor can execute the exact same instructions simultaneously. In CUDA, each block is assigned to a multiprocessor and the threads are assigned to different stream processors. Hence, blocks do not need any kind of synchronization in order to execute in parallel. This is part of the Single Instruction Multiple Threads (SIMT) architecture [16].
2.3.2 Memory hierarchy

There are three different types of memory spaces in which variables can be declared:

- **Private**: Each thread has its own private memory space. Variables in this memory space can only be read from and written to by the thread that owns it.

- **Shared**: Each block has a shared memory space. Variables in this memory space can be written to and read from by any thread within the block.

- **Global**: All threads are allowed to read from and write to variables in this memory space.
Figure 3: Address spaces.
3 Background

Based on paper [1], the authors try to find a solution for finding feasible schedules for uncertain, periodic, self-suspending real-time tasks. The idea is interesting because it has been shown to be difficult to find solutions for this task model [22]. The tasks are defined as uncertain because their execution and suspension times can vary (but they are defined within an interval of time). The tasks execute periodically without priorities. The difference from the periodic task model [11] is that the task execution is fragmented with suspension intervals which is decided by the task itself, however, the time intervals are clearly defined as part of this task model.

In essence, we believe that tools such as UPPAAL-TIMES [3] and UP-PAAL [4] do not scale with complex models when it comes to verification. It is usual that the verification runs out of memory. Conclusively, model checking tools cannot cope with complex models such as self-suspending tasks. We want to investigate if the GPU can accelerate the model checking up to the point where it is feasible to find solutions for complex models like the self-suspending task model. Tools such as UP-PAAL use an extensive amount of memory when verifying models. UPPAAL-TIMES has memory problems when verifying our model of self-suspending tasks, even on modern computers with up to 4 GB of memory.

The best possible result for this thesis would be if the GPU accelerated model checking manages to find feasible scheduling paths in the order of milli or micro seconds. In this case it would be possible for the GPU-accelerated model checker to assist an operating system scheduler with finding new (more efficient) scheduling paths during runtime.

Although many parts of this thesis (the idea of model checking self-suspending tasks, the task-set used in our verification etc.) are based on [1], this paper is actually not technically sound when it comes to the verification results. Hence, we cannot compare our results with theirs (even though we have access to their implementation). Figure 4\(^1\) shows an UPPAAL-TIMES model of the original UPPAAL-TIGA model presented in [1]. The input to this model are the execution and suspension time durations of tasks. However, note that there are three transition guards with (WL-1) or (WU-1). The variables WL and WU hold the lower and upper interval value of the task execution time and the input values WL1 and WU1 to the model correspond to the correct values. Hence, the task execution times will be 1 time-unit less than the original value. This means that the verifications will pass easily and fast because the system utilization decreases. However, removing the

\(^1\)http://igm.univ-mlv.fr/~masson/Softwares/SelfSuspending/
subtraction in \((WL-1)\) and \((WU-1)\) does not give a correct verification either because the implemented model is then incorrectly modelled. This can be observed when simulating the model. The model introduces idle time at certain time points which makes the schedule incorrect. The conclusion is that we cannot compare our results with this model even though we use the same task systems as presented in the paper.

Figure 4: UPPAAL model used in [1].
4 Related work

The most related work [1] with respect to this thesis use the tools UPPAAL and UPPAAL-TIGA [4] to find schedulable paths for two different task sets. We use the same task sets in this thesis, however, our main focus is to conduct the verification on the GPU hardware in order to speed-up the process. Statistical model checking [24] use roughly the same technique as we do in this thesis. Instead of searching the entire state-space [4], they search a part of it in order to decrease the time length. Statistical approaches are often used when the state-space is to large for systematic searches. UPPAAL-SMC [4] is based on statistical model checking and can give, for example, probabilities that certain properties are satisfied at different time instances in the model. Looking at speed-up of model checking, such work has been done [18] but the focus has been on faster searches by using distributed computers that communicate with each other in order to find good paths faster. This is, however, only relevant for a CPU version of the algorithm used in this thesis since the amount of memory needed for the search algorithms are too high for GPUs. In [8] a responsive GPGPU execution model has been developed so that tasks can execute some of their load on the GPU. The difference from our work is that we synthesize the scheduler on the GPU while [8] run the tasks on the GPU instead.
5 Problem formulation

The goal of this thesis is to investigate how efficient a NP-hard search-problem can be solved using the GPU. We have modelled a scheduler in UPPAAL-TIMES that is suited for scheduling periodic self-suspending tasks. Generated C-code from the model will represent the model that we will search. The example system that we will try to find feasible schedules for is shown in Table 1. Hence, our model implements these three tasks.

<table>
<thead>
<tr>
<th>Task</th>
<th>Period (T)</th>
<th>Deadline (D)</th>
<th>Execution pattern (P)</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ₁</td>
<td>10</td>
<td>10</td>
<td>(2,2,4)</td>
</tr>
<tr>
<td>τ₂</td>
<td>20</td>
<td>20</td>
<td>(2,8,2)</td>
</tr>
<tr>
<td>τ₃</td>
<td>12</td>
<td>12</td>
<td>(2)</td>
</tr>
</tbody>
</table>

Table 1: Task system used in our model.

Figure 5 shows the optimal schedule [1] for the taskset in Table 1. Note that neither FPS nor EDF is optimal in finding schedules for periodic self-suspending tasks. Hence, this schedule can not be derived using any scheduling algorithm.

The goal is to find out if a model-checking algorithm, running on a GPU, can find a feasible schedule for the taskset in Table 1. If it can then the resulting schedule should look like the one in Figure 5.
6 Analysis of problem

This section will describe the different problem aspects of synthesizing a model checking algorithm onto a GPU hardware platform.

The biggest constraint on a GPU is memory. This makes it impossible to save already-tested paths. The second issue is to make the entire search as parallelized as possible in order to achieve good speed-up.

6.1 Choosing a language

There are two GPU programming languages suitable for this project: OpenCL and CUDA. The biggest advantage of choosing OpenCL is that it can be run on almost any GPU, while CUDA can only be run on certain NVIDIA GPUs. The advantage of CUDA is that it is optimized for NVIDIA GPUs, so it is faster than OpenCL on these GPUs. Also, you can work with CUDA in Visual Studio 2010 which is a familiar working environment for the members in this project. Another consideration is how good the support is for debugging the code in these two languages. Last but not least, the available hardware platform, i.e. NVIDIA or AMD, will affect the choice of language as well.

6.2 Parallelization

GPUs are designed for making heavy, parallelized calculations [8]. The generated code has to be modified so that it can take advantage of this in order to make the search faster.

6.2.1 Branches

The threads in CUDA and OpenCL are divided into blocks. The threads in a block have to be on the same path in the code in order to be able to make calculations in parallel. If the threads are divided into different paths, each path will be executed serially [16]. The number of branches in the code must be kept to a minimum.

The biggest problem, considering branches in the original generated C-code, is the function that evaluates the guards in the automaton. This function will be called before a transition is taken in the automaton so that a valid transition will be taken. The function consists of a switch-case statement with one case for every transition. Since each thread represents a transition, this function will divide all of the threads to different cases and thereby slow down the program significantly. This part of the code has to
be rewritten in order for it to become more parallelized and thereby more suitable for execution on a GPU.

6.2.2 Stopping the search

Since every block works independently [16], a way has to be found for one block to tell all blocks when a successful path has been found. Otherwise, all other blocks will continue searching and the CPU will not get the result until all blocks are finished.

6.3 Memory restrictions

The GPU has a smaller amount of low-latency memory compared to the CPU [8]. The generated code has to be adjusted to fit the memory model of a GPU. Another serious problem is the memory locality, i.e., memory caching. Memory must be used in a sequence that is adapted to the GPU memory cache-algorithm in order for the code to be as efficient as possible.

6.3.1 Choosing paths

In order for the search to be fast, we must avoid repeating already taken paths. A simple way to do this is to save already taken paths to memory. Considering the memory restrictions on the GPU, this would be very inefficient. Hence, a systematic and memory efficient way to choose paths must be implemented so that paths are not repeated so often.
7 Method

7.1 Parallelization

In the original code, the function that evaluates the guards of the transitions in the automaton consists of a switch-case statement where each case represents a transition. It would be natural to let each thread represent a transition, but this would split all threads into different cases which would lead to a lot of branching. Every case in the switch-case statement consists of a number of comparison operations performed on variables and/or constants. The results from all the comparisons are added together with the binary AND operator. The transition is marked active if the result is equal to 1.

The first step towards parallelization is to add dummy comparisons to all guards to make sure that the same operations are executed on all guards and in the same order. In order for the dummy comparisons to not affect the guards, they should always be true. Here is a simplified example of how the generated code can be:

```c
int actives[3];

int eval_guard (int trn)
{
    switch (trn)
    {
        case 0: return (Clock > 3);
        case 1: return (Clock != 2);
        case 2: return (Clock < 1);
    }
}

void collect_actives()
{
    for (int i=0; i<3; i++)
        actives[i]=eval_guard(i);
}
```

For this code, every time `collect_actives()` is executed the array `actives[]` is updated with the three transitions’ statuses. We start by adding dummy comparisons for each guard:

```c
int eval_guard (int trn)
{
```
switch (trn)
{
    case 0: return (Clock > 3 && 1 != 0 && 0 < 1);
    case 1: return (1 > 0 && Clock != 2 && 0 < 1);
    case 2: return (1 > 0 && 1 != 0 && Clock < 1);
}

Every guard now has the same number of operands, so a matrix can be
made that holds those values and each thread can be assigned a transition.
Now it is every thread’s responsibility to update the status of their assigned
transition.

```
int operands[3][6] = {
    {Clock, 3, 1, 0, 0, 1},
    {1, 0, Clock, 2, 0, 1},
    {1, 0, 1, 0, Clock, 1}
};
```

```
int eval.guard(int trn)
{
    return operands[trn][0] > operands[trn][1] &&
    operands[trn][2] != operands[trn][3] &&
    operands[trn][4] < operands[trn][5];
}
```

```
void collect.actives()
{
    int trn = getthreadID();
    actives[trn]=eval.guard(trn);
}
```

The next problem was that when a variable used in the calculations was
changed, it needed to be changed everywhere in the matrix. Therefore, the
matrix was remade to hold pointers which points to the addresses of the
variables used, instead of the actual values. Now each variable only needs to
be changed in one place.

### 7.2 Seed system

In tree searching it is normal to save each path in memory to make sure that
you don’t take the same path more than once. When performing exhaustive
searches on large trees on a GPU you don’t have the luxury of being able to
save already-tested paths in memory because the blocks would have to use global memory. Global memory on a GPU is the memory with the highest latency and would probably hinder the search from being fast enough [16]. One way to search trees without saving paths is to make the entire search random. However, this does not guarantee that every leaf will be found and the expected time for the search to complete is unpredictable. Another way to perform an exhaustive search is by doing it recursively, but this will also be expensive in terms of memory. Therefore, we developed an algorithm, which to the best of our knowledge did not exist before. It tests many paths but few are repeated. This algorithm was named Seeded Tree Search (STS) and it requires only that the current seed and the current product of branches need to be stored in memory. These are both represented as regular integers.

7.2.1 Seeded Tree Search (STS)

In STS, each path is based on a number, called a seed. Information of how many branches the path has passed is kept in a variable, and it is reset to the value 1 every time a new seed is tested. At every node, the seed will be divided by the value of the branches variable. The result of this division is divided by the number of branches for the current node. The next node in the path will be chosen based on the remainder of the second division. Finally, the branches variable is updated by multiplying its current value with the number of branches for the current node.

Algorithm 1 Seeded Tree Search.

\[
\begin{align*}
\text{H} & \quad \text{seed} \leftarrow \text{start\_seed} \\
\text{while} & \quad \text{seed} < \text{end\_seed} \text{ do} \\
& \quad \text{branches} \leftarrow 1 \\
& \quad \text{node} \leftarrow \text{root} \\
& \quad \text{while} \quad \text{node} \text{ is not a leaf} \text{ do} \\
& \quad & \quad \text{choice} \leftarrow \text{seed} \\
& \quad & \quad \text{choice} \leftarrow \text{choice} \mod \text{node.branches} \\
& \quad & \quad \text{branches} \leftarrow \text{branches} \ast \text{node.branches} \\
& \quad & \quad \text{node} \leftarrow \text{node.branch} \!(\text{choice}) \\
& \quad \text{end while} \\
& \quad \text{seed} \leftarrow \text{seed} + 1 \\
\text{end while}
\end{align*}
\]

To calculate the number of seeds needed with STS to make sure all leaves are reached, find the leaf with the least probability of being reached and
multiply the number of branches for each node on the path from the root down to that node. See figure 6 for an example.

Figure 6: Amount of searches needed to cover all paths with STS is 2 \times 2 \times 3 = 12.

In figure 6 there is a tree with five leaves. To reach all the leaves with STS, twelve seeds are needed. The optimal number of paths that will reach all leaves will be five. A recursive search will accomplish this, but would require more memory than STS. This is not a problem for the tree in Figure 6, but a tree with millions of leaves or more could be an issue in terms of memory. With a fully balanced tree, STS finds all leaves without saving any paths and without taking the same path more than once (Figure 7). The reason why this algorithm works in this way is because we check at every node how many times we have been there before. This is done by using the division at line 6 in Algorithm 1.
Figure 7: STS on a balanced tree with the first seed for every path.

7.3 Search model

7.3.1 CUDA

CUDA C was chosen as language for this thesis. This was mainly because of the available hardware, i.e. NVIDIA. Code written in CUDA C is divided into host code and device code. The host code is regular C code with different calls to CUDA functions, i.e. to allocate memory on the GPU. The device code is executed on the GPU and has some extra functionality on top of regular C. (This functionality is there to be able to specify in which address space different variables should be stored and also to be able to synchronize threads in different ways). The host code has the entry point for the program and starts the device code by calling a GPU kernel function. When the call is made, the host code specifies the number of threads and blocks. Each block will have its own unique seed numbers and it is responsible for testing if any of them succeeds in finding a feasible schedule. Every block must have at least as many threads as there are transitions in the automaton.

7.3.2 Restrictions

During testing, it was observed that GPUs can only run for a certain amount of time. If that time is exceeded then the GPU reboots. Therefore, a search can not go on forever, so the CPU has to perform sub-searches on the GPU in intervals. In our implementation, a loop is executed on the CPU that transfers new seed numbers to the GPU at every iteration. If a successful result is returned from the GPU then the loop exits.
7.3.3 Device code

7.3.3.1 Memory All the variables that are used by all transitions must be in shared address space which means that all threads in a block will be able to reach them. The matrix of operands is represented as a list of private pointers for each thread, pointing to different shared variables. Seed numbers and results are represented as global arrays. The seed numbers are copied to the device before starting a sub-search and the results are copied to the host when a sub-search is finished. The global result array is also used by one block to notify other blocks that a sub-search has been successful.

7.3.3.2 Layout The device code starts by assigning one thread to be the coordinator for the sub-search. It is followed by letting the coordinator set up the entire automaton with shared variables. When this is done, the sub-search is started by going into a loop. The first job in the loop is to check if the sub-search must be reset with a new seed number, which would happen if the previous seed failed. If this is the case, then the coordinator resets all the shared variables and starts over with a new seed number. The second job for the coordinator is to update any shared variables that is in need of an update. After the update, the non-coordinating threads update their transition status which is located in shared address space. After this, the coordinator goes through the results to sort out which transitions are active and makes a choice based on the current seed number. Finally, the coordinator makes the selected transition. At the end of the loop, every thread checks if the path was successful and exits the loop if that was the case. When a sub-search is successful and the loop has stopped, then the coordinator for that block provides the global result array with the results so every block observes it and stops.

7.3.4 Host code

The host code is responsible for allocating memory on the GPU device that will hold the seed numbers and the result. Once the allocation is done, then the seed numbers are copied to the device and sub-searches are started by calling the GPU kernel function. When the sub-searches are finished, then the result is copied back to the host for evaluation. If the result indicates that the sub-searches have failed, then new sub-searches with new seed numbers will be started.
8 Solution

The implementation of this model consists of converting regular C code, generated by UPPAAL-TIMES from the timed automaton, to CUDA C code. This is done with the following steps within the GPU kernel function.

1. Map all the private pointers so that the threads point to the correct variables in the switch-case statement in the original code.

2. Let one thread set all the variables for the automaton. This thread is called the coordinator.

3. Create a while-loop and make it do the following at every iteration. If the search is in need of a reset, then let the coordinator reset the entire automaton and increase the seed number.
   (a) Let the coordinator update any pointers that must point to a new address.
   (b) Let every transition-thread update the status of their corresponding transition.
   (c) Let the coordinator collect all active transitions.
   (d) Let the coordinator choose one of the active transitions based on the current seed number. If no transitions are active, then increase the system time.
   (e) Let the coordinator check that an endless transition loop has not occurred. If it has, then reset with a new seed number.
   (f) Let the coordinator check if a successful state has been reached, and if so, notify other threads within the block and exit the loop.

4. Let the coordinator check if a successful state has been reached and provide the global result array with the result if this is the case. There must also be a test to see if any other block has already reached a successful state. If that is the case then no modification will be made to the result array.

The following steps are made in the host code:

1. Allocate memory on the host for the seed-number array and the result array.

2. Allocate memory on the device for the seed-number array and the result array.
3. Create a while-loop with a condition that is true until a successful state has been found.

(a) Set all result numbers in the result array on the device to zero.

(b) Fill the seed-number array on the host with appropriate values. For a linear search this should start with zero. The seed numbers should be randomized in a random search.

(c) Copy the content of the seed-number array on the host to the seed-number array on the device.

(d) Run the GPU kernel function with the desired amount of blocks and threads. The number of threads in each block should be at least the same amount as the number of transitions in the automaton.

(e) Copy the content of the result array on the device to the result array on the host.

(f) Evaluate the result array to see if a successful state has been found. If so, exit the loop.

4. Present the successful path.

8.1 Modified STS

Due to the nature of the automaton, there is a risk that it might end up in an endless transition-loop. Therefore, the Seeded Tree Search was slightly modified to cope with this. Instead of always dividing the current seed number with the number of branches, a condition was created to check if the number of branches has exceeded the seed number. If this is true, then the branches variable will stop increasing and the chosen branch will be based on the remainder of the seed number divided by the number of branches for the current node. It was obvious during our tests that this algorithm was more efficient.
Algorithm 2 Seaded Tree Search (Modified).

\[
\begin{align*}
\text{seed} &\leftarrow \text{startseed} \\
\text{while} \ \text{seed} < \text{endseed} \ \text{do} \\
& \quad \text{branches} \leftarrow 1 \\
& \quad \text{node} \leftarrow \text{root} \\
& \quad \text{while} \ \text{node} \ \text{is not a leaf} \ \text{do} \\
& \quad \quad \text{if} \ \text{seed} < \text{branches} \ \text{then} \\
& \quad \quad \quad \text{choice} \leftarrow \text{seed} \ \text{mod} \ \text{branches} \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{choice} \leftarrow \text{seed} \\
& \quad \quad \quad \text{choice} \leftarrow \text{choice} \ \text{mod} \ \text{node.branches} \\
& \quad \quad \quad \text{branches} \leftarrow \text{branches} \ \times \ \text{node.branches} \\
& \quad \quad \text{end if} \\
& \quad \quad \text{node} \leftarrow \text{node.branch(\text{choice})} \\
& \quad \text{end while} \\
& \quad \text{seed} \leftarrow \text{seed} + 1 \\
& \text{end while}
\end{align*}
\]

8.2 Example of converting guard evaluations from C to CUDA C

The example automaton consists of five transitions. Part of the original switch-case statement for this example is shown below:

\[
\begin{align*}
// \text{evaluate guards} \\
\text{switch (trn)} \\
&\{ \\
&\quad \text{case 0: return (Sched_i!=NrTasks);} \\
&\quad \text{case 1: return (Sched_i<NrTasks);} \\
&\quad \text{case 2: return (Sched_i==NrTasks);} \\
&\quad \text{case 3: return (NrTasks<2);} \\
&\quad \text{case 4: return (Sched.ReleaseQ[Sched_i]<=Sched.ReleaseQ[Sched_i+1]} \\
&\quad \quad \quad && \& \ Sched.k!=(\text{NrTasks}−1));
&\}
\end{align*}
\]

If this code is executed in CUDA and trn is set to thread ID, then each thread will branch to its own case. This means that none of the threads will be able to evaluate the guards simultaneously [16]. First, thread 0 will evaluate case 0, and then thread 1 will evaluate case 1 etc. In order to make the threads evaluate the guards simultaneously in CUDA, the following setup must be done.
// _shared__ lets all threads in a block share variables
_shared__ int actives[5];

// dummy variables and variables that needs updating
_shared__ int zero, one, two, Sched_ReleaseQ_Sched_i,
    Sched_ReleaseQ_Sched_i_plus_one, NrTasks_minus_one;

// private for each thread
int *operand0, *operand1, *operand2, *operand3,
    *operand4, *operand5, *operand6, *operand7;
int threadID, coordinator;

// figure out threadID for each thread
threadID = threadIdx.y * blockDim.x + threadIdx.x;

// let thread 0 be the coordinator
coordinator = 0;

// only the coordinator sets shared variables
if (threadID == coordinator)
{
    zero = 0;
    one = 1;
    two = 2;
    NrTasks_minus_one = NrTasks - 1;

    // these two needs to keep updating
    Sched_ReleaseQ_Sched_i = Sched_ReleaseQ[Sched_i];
    Sched_ReleaseQ_Sched_i_plus_one = Sched_ReleaseQ[Sched_i+1];
}

// have every thread set its own private pointers
if (threadID == 0)
{
    operand0 = &Sched_i;  // !=
    operand1 = &NrTasks;  // !=
    operand2 = &zero;     // <
    operand3 = &one;      // <
    operand4 = &one;      // ==
    operand5 = &one;      // ==
    operand6 = &one;      // <=
operand7 = &one;  // <=

if (threadID == 1)
{
    operand0 = &zero;  // !=
    operand1 = &one;  // !=
    operand2 = &Sched_i;  // <
    operand3 = &NrTasks;  // <
    operand4 = &one;  // ==
    operand5 = &one;  // ==
    operand6 = &one;  // <=
    operand7 = &one;  // <=
}

if (threadID == 2)
{
    operand0 = &zero;  // !=
    operand1 = &one;  // !=
    operand2 = &zero;  // <
    operand3 = &one;  // <
    operand4 = &Sched_i;  // ==
    operand5 = &NrTasks;  // ==
    operand6 = &one;  // <=
    operand7 = &one;  // <=
}

if (threadID == 3)
{
    operand0 = &zero;  // !=
    operand1 = &one;  // !=
    operand2 = &NrTasks;  // <
    operand3 = &two;  // <
    operand4 = &one;  // ==
    operand5 = &one;  // ==
    operand6 = &one;  // <=
    operand7 = &one;  // <=
}

if (threadID == 4)
{
    operand0 = &Sched_k;  // !=
    operand1 = &NrTasks_minus_one;  // !=
operand2 = &zero;  // <
operand3 = &one;    // <
operand4 = &one;  // ==
operand5 = &one; // ==
operand6 = &Sched_ReleaseQ_Sched_i;  // <=
operand7 = &Sched_ReleaseQ_Sched_i_plus_one;  // <=
}

The next step is to update the pointers and then let every thread perform its own evaluation simultaneously with the other threads.

// coordinator updates values from arrays indexed with variables
if (threadID == coordinator)
{
    Sched_ReleaseQ_Sched_i = Sched_ReleaseQ[Sched_i];
    Sched_ReleaseQ_Sched_i_plus_one = Sched_ReleaseQ[Sched_i+1];
}

// all threads (transitions) update their status (evaluate guard)
actives[threadID] =  (*operand0 != *operand1) &
                    (*operand2 < *operand3) &
                    (*operand4 == *operand5) &
                    (*operand6 <= *operand7);

9 Results

9.1 Experimental setup

Our experiments were performed on two different GPUs, an NVIDIA GeForce GT 320M with 24 CUDA cores running at 450 MHz and an NVIDIA GeForce GTX 275 with 240 CUDA cores running at 633 MHz.

An important part of testing the model is to make sure how many blocks the GPU can handle. Due to the SIMT Architecture mentioned in Section 2.3.1, the number of blocks should never exceed the number of multi-processors on the GPU and the number of transitions in the automaton should never exceed the maximum amount of threads per block on the GPU. The GT 320M GPU has 3 multi-processors so only 3 blocks could be used, while the GTX 275 GPU has 30 multi-processors allowing us to have up to 30 blocks. Another important aspect is the time that the GPU is allowed to run before it reboots. This is different from one GPU to another. We will elaborate more on this in Section 9.3.

The experiments in this thesis consist of two different setups; one random and one linear search. The linear search is always started on seed number zero and progresses through all seed numbers linearly until either a successful result is found or until the search reaches an error state. The random search was performed with each block starting on a random seed number. It then continues with a linear search until the time-limit for the GPU is reached or until a successful result is found. If the time-limit of the GPU is reached, then new random seed numbers are generated and the blocks restart their linear search.

9.2 Finding feasible scheduling paths

These experiments were conducted using both the random and the linear search method. With the random search method a successful path was found up to time unit 20 in the scheduling graph in Figure 5. However, the linear search method was only successful up to time unit 12. We did 10 test rounds for every time unit from 1 up to the highest reachable time unit with both search methods. We measured the time it took to find a successful path for each time unit in the scheduling graph.
Figure 8: Search-time results up to 10 time units in the scheduling graph.

Figure 9: Search-time results between 11 and 12 time units in the scheduling graph.
Figures 8, 9 and 10 show the average time it took to find feasible scheduling paths for time unit 1 up to 20 in the scheduling graph in Figure 5. We can observe that the linear search was unable to find paths, in a timely manner, from time unit 12 and upwards. For more details, see Table 3, 4, 5 and 6 in Appendix.

Figure 11 shows a schedule up to time unit 15 that our GPU model-checker managed to find. Our CUDA program sends a unique path ID back to the CPU if a scheduling path is found. This ID can then be decoded by a corresponding C-program of the model on the CPU. During decoding, the C-program logs the schedule, and it finishes by creating a Grasp schedule like the one in Figure 11.
9 RESULTS

9.3 Paths per second

We also did experiments to find out how many paths per second we could traverse with the two GPU cards. In addition, we also compared the GPU results with two other CPU platforms:

- AMD Opteron (16 core) 2.0 GHz CPU
- Intel Core2 Duo (2 core) P9400 2.4 GHz CPU

Table 2 shows the measured (average) paths per second in the timed automaton for these four platforms. The GT 320M GPU could run for almost twenty seconds straight before it needed to reboot, while the GTX 275 GPU could only run for about one second. It is interesting to note that cheap and outdated GPUs match (or even outperforms) more modern (and expensive) CPUs in terms of paths per second. The GT 320M, GTX 275 and the Intel Core2 Duo date back to 2008/2009 while the AMD Opteron represents the state-of-the-art in hardware. The CPU version of our STS search algorithm were, in these experiments, parallelized using the OpenMP library. Hence, the algorithm utilized 2 CPU cores on the Intel Core2 Duo machine and it used 16 cores on the AMD Opteron platform.

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Nr of paths/second</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA GeForce GTX 275</td>
<td>4875</td>
</tr>
<tr>
<td>AMD Opteron</td>
<td>350</td>
</tr>
<tr>
<td>NVIDIA GeForce GT 320M</td>
<td>300</td>
</tr>
<tr>
<td>Intel Core2 Duo</td>
<td>285</td>
</tr>
</tbody>
</table>

Table 2: Paths per second on different hardware platforms.

Figure 12 shows the benchmark results from G3D Mark\(^2\). This benchmark measures how efficient the graphics cards can render, display and play movie scenes. We can see that the GTX 275 and GT 320M card are not even close to performance compared to newer cards such as the GTX 580. A note to the reader is that the newest GTX card (GTX 590) has twice the amount of CUDA cores (1024 cores) compared to GTX 580 and the NVIDIA Tesla series offer even more computing power (up to 3000 cores).

\(^2\)http://www.videocardbenchmark.net/gpu.php?gpu=GeForce+GT+320M
Figure 12: G3D Mark benchmark-results for various graphics cards.
10 Conclusion

The random search method was more effective than the linear one. It would probably be even faster if there was a fast enough way to randomize numbers in the GPU kernel code so the program could take fully randomized paths, instead of just randomizing the seeds. This would prevent the automaton from getting stuck in endless transition loops.

The results reached in this project do not show that it is currently possible to use GPU model checking for online scheduling of self-suspending real-time tasks. However, optimized code combined with a high-end GPU card (with more CUDA cores) together with more complex search techniques could perhaps make it possible. Since the search-problem is NP-hard it could be interesting to look at new search methods using heuristics.

However, the speed we achieved (in terms of searches per second) with low-end GPUs were surprisingly good compared to CPUs. Conclusively, GPU model checking could be useful in verification purposes if more powerful hardware was used with efficient search methods.

An interesting finding from this project is that the GPU (and the corresponding CPU version) model-checker managed to find scheduling solutions up to time unit 20 in the scheduling graph. UPPAAL-TIMES could not find any scheduling path in the corresponding model.
11 Future work

There are several improvements that could be done in order to get more efficient model-checking using the GPU. First of all, the generated C-code from UPPAAL-TIMES could be optimized so that fewer CUDA cores would be needed to check guards in transitions. This would increase the number of work-groups and hence increase the number of searches per second. The second improvement would be to try out state-of-the-art GPUs such as Nvidia GeForce GTX 590 or Nvidia Tesla K10. The third and most important improvement would be to try out different search techniques.
References


REFERENCES


Appendix

Tables 3, 4, 5 and 6 show the measured time it took to find feasible schedules. Rows represent the time units in the scheduling graph. The columns represent the test results for each round and the last column presents the average results from all 10 rounds. The presented results are represented in milliseconds and the best result(s) for each time unit is written in bold font.

Table 3: Results for random search with GT320M GPU.

<table>
<thead>
<tr>
<th>TU</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.7</td>
<td>4.5</td>
<td>3.7</td>
<td>4.6</td>
<td>3.7</td>
<td>4.5</td>
<td><strong>3.6</strong></td>
<td>4.5</td>
<td>3.7</td>
<td>4.5</td>
<td>4.1</td>
</tr>
<tr>
<td>2</td>
<td>5.1</td>
<td>4.9</td>
<td>5.2</td>
<td>6</td>
<td>6.9</td>
<td>4.9</td>
<td>6</td>
<td><strong>4.1</strong></td>
<td>4.1</td>
<td>9</td>
<td>5.62</td>
</tr>
<tr>
<td>3</td>
<td>5.6</td>
<td>9.2</td>
<td>5.7</td>
<td>6.2</td>
<td>5.5</td>
<td>6.9</td>
<td>5.5</td>
<td>9.4</td>
<td><strong>5.4</strong></td>
<td>8.3</td>
<td>6.77</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>8.3</td>
<td>12.6</td>
<td>11.2</td>
<td>7.1</td>
<td><strong>6.3</strong></td>
<td>11.2</td>
<td>25.1</td>
<td>7.8</td>
<td>10.8</td>
<td>11.24</td>
</tr>
<tr>
<td>5</td>
<td>8.7</td>
<td>25.2</td>
<td>9.1</td>
<td>8.5</td>
<td>8</td>
<td>8</td>
<td>12.5</td>
<td>14</td>
<td>13.1</td>
<td><strong>5.3</strong></td>
<td>11.24</td>
</tr>
<tr>
<td>6</td>
<td>6.7</td>
<td>23.9</td>
<td>14.6</td>
<td>11.9</td>
<td>9.1</td>
<td><strong>6.6</strong></td>
<td>12.9</td>
<td>11.8</td>
<td>13.5</td>
<td>16.9</td>
<td>12.79</td>
</tr>
<tr>
<td>7</td>
<td>13.1</td>
<td>13.7</td>
<td>24.6</td>
<td><strong>9.4</strong></td>
<td>16.1</td>
<td>12.6</td>
<td>19.9</td>
<td>14.2</td>
<td>13</td>
<td>14.3</td>
<td>15.09</td>
</tr>
<tr>
<td>8</td>
<td>24.7</td>
<td>11.7</td>
<td>16.7</td>
<td>13.9</td>
<td>17</td>
<td>25.5</td>
<td><strong>7.4</strong></td>
<td>12</td>
<td>29.9</td>
<td>26.9</td>
<td>16.87</td>
</tr>
<tr>
<td>9</td>
<td>23.9</td>
<td>27.9</td>
<td>11.3</td>
<td><strong>9.6</strong></td>
<td>25.5</td>
<td>15.8</td>
<td>14.3</td>
<td>26.7</td>
<td>15.2</td>
<td>15.7</td>
<td>18.59</td>
</tr>
<tr>
<td>10</td>
<td>39.8</td>
<td>15</td>
<td>22.6</td>
<td>15</td>
<td>25.5</td>
<td>48.3</td>
<td><strong>12.3</strong></td>
<td>17.6</td>
<td>17.2</td>
<td>26.7</td>
<td>24</td>
</tr>
<tr>
<td>11</td>
<td>1670</td>
<td>5205</td>
<td>550</td>
<td>2885</td>
<td>8751</td>
<td>3581</td>
<td>29778</td>
<td>921</td>
<td>3814</td>
<td><strong>278</strong></td>
<td>5743.3</td>
</tr>
<tr>
<td>12</td>
<td>11864</td>
<td>16589</td>
<td>5913</td>
<td>17044</td>
<td>8970</td>
<td>28650</td>
<td>5866</td>
<td>7183</td>
<td><strong>578</strong></td>
<td>14719</td>
<td>11737.6</td>
</tr>
<tr>
<td>13</td>
<td>74928</td>
<td>409</td>
<td>7111</td>
<td>1078</td>
<td>21784</td>
<td>43096</td>
<td>1054</td>
<td>32118</td>
<td>17534</td>
<td>18181</td>
<td>21729.3</td>
</tr>
<tr>
<td>14</td>
<td><strong>6074</strong></td>
<td>19422</td>
<td>35317</td>
<td>11606</td>
<td>25971</td>
<td>26302</td>
<td>15573</td>
<td>34306</td>
<td>19027</td>
<td>13861</td>
<td>20745.9</td>
</tr>
<tr>
<td>15</td>
<td>19016</td>
<td>121718</td>
<td>58485</td>
<td>18790</td>
<td>1468</td>
<td>5177</td>
<td><strong>875</strong></td>
<td>10769</td>
<td>19934</td>
<td>27498</td>
<td>28373</td>
</tr>
<tr>
<td>16</td>
<td><strong>6665</strong></td>
<td>8984</td>
<td>46871</td>
<td>21690</td>
<td>10745</td>
<td>56228</td>
<td>41493</td>
<td>26598</td>
<td>26125</td>
<td>54023</td>
<td>29942.2</td>
</tr>
<tr>
<td>17</td>
<td>16140</td>
<td>4541</td>
<td>79927</td>
<td>18013</td>
<td><strong>2086</strong></td>
<td>46662</td>
<td>19005</td>
<td>5041</td>
<td>75177</td>
<td>89623</td>
<td>35755.5</td>
</tr>
<tr>
<td>18</td>
<td>70579</td>
<td>39524</td>
<td>18424</td>
<td><strong>8248</strong></td>
<td>44133</td>
<td>74074</td>
<td>149756</td>
<td>77567</td>
<td>71265</td>
<td>36494</td>
<td>58628.3</td>
</tr>
<tr>
<td>19</td>
<td>89315</td>
<td>27159</td>
<td>26428</td>
<td><strong>288</strong></td>
<td>21788</td>
<td>21244</td>
<td>22823</td>
<td>16550</td>
<td>53313</td>
<td>84995</td>
<td>35490.3</td>
</tr>
<tr>
<td>20</td>
<td>21375</td>
<td>1237</td>
<td><strong>377</strong></td>
<td>7276</td>
<td>54038</td>
<td>51608</td>
<td>10757</td>
<td>38180</td>
<td>114371</td>
<td>8608</td>
<td>30802.7</td>
</tr>
</tbody>
</table>

Table 4: Results for linear search with GT320M GPU.

<table>
<thead>
<tr>
<th>TU</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.7</td>
<td><strong>3.7</strong></td>
<td>4.5</td>
<td>4.6</td>
<td>4.7</td>
<td>4.5</td>
<td>4.4</td>
<td>4.4</td>
<td>4.4</td>
<td>4.4</td>
<td>4.51</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>4.8</td>
<td>4.9</td>
<td>5</td>
<td>5.7</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>4.1</td>
<td>4</td>
<td>4.86</td>
</tr>
<tr>
<td>3</td>
<td>5.4</td>
<td>5.3</td>
<td>5.3</td>
<td>5.3</td>
<td>5.3</td>
<td>5.1</td>
<td>5.3</td>
<td>5.4</td>
<td><strong>4.3</strong></td>
<td>5.4</td>
<td>5.21</td>
</tr>
<tr>
<td>4</td>
<td><strong>4.7</strong></td>
<td>4.8</td>
<td>4.9</td>
<td>5.6</td>
<td>5.6</td>
<td>5.6</td>
<td>4.8</td>
<td>6.1</td>
<td>6.7</td>
<td>5.6</td>
<td>5.44</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6.2</td>
<td>6.1</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td><strong>5.8</strong></td>
<td>6.1</td>
<td>6.04</td>
</tr>
<tr>
<td>6</td>
<td>6.3</td>
<td>5.7</td>
<td>5.7</td>
<td>6.4</td>
<td>6.6</td>
<td>6.5</td>
<td><strong>5.6</strong></td>
<td>6.3</td>
<td>5.7</td>
<td>6.5</td>
<td>6.13</td>
</tr>
<tr>
<td>7</td>
<td>6.8</td>
<td>6.7</td>
<td>6.8</td>
<td>6</td>
<td>6.8</td>
<td>6.8</td>
<td><strong>6</strong></td>
<td>6.9</td>
<td>7</td>
<td>6</td>
<td>6.58</td>
</tr>
<tr>
<td>8</td>
<td>7.1</td>
<td><strong>6.3</strong></td>
<td>6.4</td>
<td>7.2</td>
<td>7.4</td>
<td><strong>6.3</strong></td>
<td>7.2</td>
<td>6.4</td>
<td><strong>6.3</strong></td>
<td>7.2</td>
<td>6.78</td>
</tr>
<tr>
<td>9</td>
<td>7.7</td>
<td>7.6</td>
<td>7.7</td>
<td>6.9</td>
<td>7.8</td>
<td><strong>6.7</strong></td>
<td>7.9</td>
<td>7.4</td>
<td>6.8</td>
<td>7.6</td>
<td>7.41</td>
</tr>
<tr>
<td>10</td>
<td>7.9</td>
<td>7.2</td>
<td>7.2</td>
<td><strong>7.1</strong></td>
<td>8</td>
<td>8</td>
<td>7.2</td>
<td>8.1</td>
<td>8.2</td>
<td>8</td>
<td>7.69</td>
</tr>
<tr>
<td>11</td>
<td>7314</td>
<td>7315</td>
<td><strong>7314</strong></td>
<td>7315</td>
<td>7314</td>
<td>7314</td>
<td>7314</td>
<td>7314</td>
<td>7314</td>
<td>7314</td>
<td>7314.5</td>
</tr>
<tr>
<td>12</td>
<td>14934</td>
<td>14936</td>
<td>14937</td>
<td>14936</td>
<td>14935</td>
<td>14936</td>
<td>14936</td>
<td>14940</td>
<td><strong>14933</strong></td>
<td>14938</td>
<td>14936</td>
</tr>
</tbody>
</table>
Table 5: Results for random search with GTX275 GPU.

<table>
<thead>
<tr>
<th>TU</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.1</td>
<td>5.1</td>
<td>2.8</td>
<td>3.8</td>
<td>2.8</td>
<td>2.8</td>
<td>3.1</td>
<td>4.5</td>
<td>4.8</td>
<td>2.9</td>
<td>3.23</td>
</tr>
<tr>
<td>2</td>
<td>3.1</td>
<td>3.1</td>
<td>2.9</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td>3.6</td>
</tr>
<tr>
<td>3</td>
<td>4.4</td>
<td>3.5</td>
<td>6.4</td>
<td>5.6</td>
<td>5.6</td>
<td>5.6</td>
<td>5.6</td>
<td>5.6</td>
<td>5.6</td>
<td>5.6</td>
<td>3.62</td>
</tr>
<tr>
<td>4</td>
<td>3.8</td>
<td>3.8</td>
<td>3.8</td>
<td>4.6</td>
<td>3.8</td>
<td>3.8</td>
<td>3.8</td>
<td>3.8</td>
<td>3.8</td>
<td>3.8</td>
<td>4.08</td>
</tr>
<tr>
<td>5</td>
<td>5.1</td>
<td>4.8</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
<td>4.2</td>
<td>4.1</td>
<td>4.1</td>
<td>4.1</td>
<td>4.62</td>
</tr>
<tr>
<td>6</td>
<td>5.1</td>
<td>4.4</td>
<td>4.4</td>
<td>4.4</td>
<td>4.4</td>
<td>4.4</td>
<td>4.4</td>
<td>4.4</td>
<td>4.4</td>
<td>4.4</td>
<td>5.55</td>
</tr>
<tr>
<td>7</td>
<td>7.8</td>
<td>6.4</td>
<td>6.4</td>
<td>6.4</td>
<td>6.4</td>
<td>6.4</td>
<td>6.4</td>
<td>6.4</td>
<td>6.4</td>
<td>6.4</td>
<td>7.22</td>
</tr>
<tr>
<td>8</td>
<td>8.1</td>
<td>5.1</td>
<td>6.9</td>
<td>10.9</td>
<td>6.9</td>
<td>10.1</td>
<td>7</td>
<td>8.7</td>
<td>6.9</td>
<td>8.2</td>
<td>8.33</td>
</tr>
<tr>
<td>9</td>
<td>8.8</td>
<td>7.2</td>
<td>8.6</td>
<td>10.9</td>
<td>6.9</td>
<td>10.1</td>
<td>7</td>
<td>8.7</td>
<td>6.9</td>
<td>8.2</td>
<td>8.33</td>
</tr>
<tr>
<td>10</td>
<td>8.1</td>
<td>8.3</td>
<td>8.8</td>
<td>9.6</td>
<td>10.5</td>
<td>11.8</td>
<td>8.6</td>
<td>9</td>
<td>11.4</td>
<td>10.7</td>
<td>9.68</td>
</tr>
<tr>
<td>11</td>
<td>5.85</td>
<td>62</td>
<td>402</td>
<td>532</td>
<td>64</td>
<td>66</td>
<td>352</td>
<td>142</td>
<td>33</td>
<td>518</td>
<td>275.6</td>
</tr>
<tr>
<td>12</td>
<td>64</td>
<td>273</td>
<td>192</td>
<td>6.1</td>
<td>114</td>
<td>353</td>
<td>146</td>
<td>241</td>
<td>989</td>
<td>1127</td>
<td>351.5</td>
</tr>
<tr>
<td>13</td>
<td>2.229</td>
<td>1812</td>
<td>389</td>
<td>3090</td>
<td>206</td>
<td>36</td>
<td>25</td>
<td>2260</td>
<td>1171</td>
<td>7964</td>
<td>1918.2</td>
</tr>
<tr>
<td>14</td>
<td>1822</td>
<td>1154</td>
<td>154</td>
<td>64</td>
<td>1321</td>
<td>319</td>
<td>459</td>
<td>28</td>
<td>1280</td>
<td>79</td>
<td>668</td>
</tr>
<tr>
<td>15</td>
<td>3006</td>
<td>1339</td>
<td>2658</td>
<td>145</td>
<td>3417</td>
<td>291</td>
<td>3754</td>
<td>859</td>
<td>2468</td>
<td>344</td>
<td>1828.1</td>
</tr>
<tr>
<td>16</td>
<td>139</td>
<td>3190</td>
<td>1257</td>
<td>213</td>
<td>2015</td>
<td>1558</td>
<td>50</td>
<td>2413</td>
<td>466</td>
<td>1448</td>
<td>1274.9</td>
</tr>
<tr>
<td>17</td>
<td>966</td>
<td>95</td>
<td>1041</td>
<td>2574</td>
<td>995</td>
<td>650</td>
<td>771</td>
<td>416</td>
<td>196</td>
<td>49</td>
<td>775.3</td>
</tr>
<tr>
<td>18</td>
<td>3499</td>
<td>1371</td>
<td>58</td>
<td>4367</td>
<td>2148</td>
<td>869</td>
<td>1393</td>
<td>1251</td>
<td>111</td>
<td>366</td>
<td>1543.3</td>
</tr>
<tr>
<td>19</td>
<td>521</td>
<td>32</td>
<td>87</td>
<td>4908</td>
<td>3334</td>
<td>1066</td>
<td>430</td>
<td>516</td>
<td>2501</td>
<td>1731</td>
<td>1512.6</td>
</tr>
<tr>
<td>20</td>
<td>2450</td>
<td>3088</td>
<td>352</td>
<td>1010</td>
<td>1282</td>
<td>377</td>
<td>901</td>
<td>767</td>
<td>494</td>
<td>2902</td>
<td>1362.3</td>
</tr>
</tbody>
</table>

Table 6: Results for linear search with GTX275 GPU.

<table>
<thead>
<tr>
<th>TU</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.6</td>
<td>3.7</td>
<td>2.8</td>
<td>3.7</td>
<td>2.8</td>
<td>3.7</td>
<td>4.7</td>
<td>3.7</td>
<td>3.7</td>
<td>3.7</td>
<td>3.8</td>
</tr>
<tr>
<td>2</td>
<td>3.1</td>
<td>3.1</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td>3.2</td>
<td>3.16</td>
</tr>
<tr>
<td>3</td>
<td>3.4</td>
<td>3.4</td>
<td>3.3</td>
<td>3.4</td>
<td>3.4</td>
<td>3.4</td>
<td>3.4</td>
<td>3.4</td>
<td>3.4</td>
<td>3.4</td>
<td>3.59</td>
</tr>
<tr>
<td>4</td>
<td>4.4</td>
<td>3.7</td>
<td>3.7</td>
<td>5.4</td>
<td>3.7</td>
<td>3.7</td>
<td>5.4</td>
<td>3.7</td>
<td>3.7</td>
<td>3.7</td>
<td>3.8</td>
</tr>
<tr>
<td>5</td>
<td>4.1</td>
<td>4.6</td>
<td>4.1</td>
<td>4.6</td>
<td>4.6</td>
<td>4.6</td>
<td>4.6</td>
<td>4.6</td>
<td>4.6</td>
<td>4.6</td>
<td>3.13</td>
</tr>
<tr>
<td>6</td>
<td>4.3</td>
<td>4.7</td>
<td>4.3</td>
<td>4.7</td>
<td>4.7</td>
<td>4.7</td>
<td>4.7</td>
<td>4.7</td>
<td>4.7</td>
<td>4.7</td>
<td>3.13</td>
</tr>
<tr>
<td>7</td>
<td>4.7</td>
<td>4.9</td>
<td>4.9</td>
<td>4.9</td>
<td>4.9</td>
<td>4.9</td>
<td>4.9</td>
<td>4.9</td>
<td>4.9</td>
<td>4.9</td>
<td>3.59</td>
</tr>
<tr>
<td>8</td>
<td>4.9</td>
<td>5.2</td>
<td>5.2</td>
<td>5.2</td>
<td>5.2</td>
<td>5.2</td>
<td>5.2</td>
<td>5.2</td>
<td>5.2</td>
<td>5.2</td>
<td>3.16</td>
</tr>
<tr>
<td>9</td>
<td>4.9</td>
<td>5.6</td>
<td>5.6</td>
<td>5.6</td>
<td>5.6</td>
<td>5.6</td>
<td>5.6</td>
<td>5.6</td>
<td>5.6</td>
<td>5.6</td>
<td>5.68</td>
</tr>
<tr>
<td>10</td>
<td>714</td>
<td>714</td>
<td>714</td>
<td>714</td>
<td>714</td>
<td>714</td>
<td>714</td>
<td>714</td>
<td>714</td>
<td>714</td>
<td>714.4</td>
</tr>
<tr>
<td>11</td>
<td>1418</td>
<td>1431</td>
<td>1418</td>
<td>1431</td>
<td>1418</td>
<td>1431</td>
<td>1418</td>
<td>1431</td>
<td>1418</td>
<td>1418</td>
<td>1418.2</td>
</tr>
<tr>
<td>12</td>
<td>1428</td>
<td>1432</td>
<td>1431</td>
<td>1431</td>
<td>1431</td>
<td>1431</td>
<td>1431</td>
<td>1431</td>
<td>1431</td>
<td>1431</td>
<td>1431.8</td>
</tr>
</tbody>
</table>