Large-Scale Parallel State Space Search Utilizing Graphics Processing Units and Solid State Disks

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund

an der Fakultät für Informatik

von

Damian Sulewski

Dortmund

2011

Tag der mündlichen Prüfung:	
Dekanin:	Prof. Dr. Gabriele Kern-Isberner
Gutachter:	Prof. Dr. Stefan Edelkamp
	Prof. Dr. Bernhard Steffen

Gewidmet meiner Familie

Abstract

The evolution of science is a double-track process composed of theoretical insights on the one hand and practical inventions on the other one. While in most cases new theoretical insights motivate hardware developers to produce systems following the theory, in some cases the shown hardware solutions force theoretical research to forecast the results to expect.

Progress in computer science rely on two aspects, processing information and storing it. Improving one side without touching the other will evidently impose new problems without producing a real alternative solution to the problem. While decreasing the time to solve a challenge may provide a solution to long term problems it will fail in solving problems which require much storage. In contrast, increasing the available amount of space for information storage will definitively allow harder problems to be solved by offering enough time.

This work studies two recent developments in the hardware to utilize them in the domain of graph searching. The trend to discontinue information storage on magnetic disks and use electronic media instead and the tendency to parallelize the computation to speed up information processing are analyzed.

Storing information on rotating magnetic disk has become the standard way since a couple of years and has reached a point where the storage capacity can be seen as infinite due to the possibility of adding new drives instantly with low costs. However, while the possible storage capacity increases every year, the transferring speed does not. At the beginning of this work, solid state media appeared on the market, slowly suppressing hard disks in speed demanding applications. Today, when finishing this work solid state drives are replacing magnetic disks in mobile computing, and computing centers use them as caching media to increase information retrieving speed. The reason is the huge advantage in random access where the speed does not drop so significantly as with magnetic drives.

While storing and retrieving huge amounts of information is one side of the medal, the other one is the processing speed. Here the trend from increasing the clock frequency of single processors stagnated in 2006 and the manufacturers started to combine multiple cores in one processor. While a CPU is a general purpose processor the manufacturers of graphics processing units (GPUs) encounter the challenge to perform the same computation for a large number of image points. Here, a parallelization offers huge advantages, so modern graphics cards have evolved to highly parallel computing instances with several hundreds of cores. The challenge is to utilize these processors in other domains than graphics processing.

One of the vastly used tasks in computer science is search. Not only disciplines with an obvious search but also in software testing searching a graph is the crucial aspect. Strategies which enable to examine larger graphs, be it by reducing the number of considered nodes or by increasing the searching speed, have to be developed to battle the rising challenges. This work enhances searching in multiple scientific domains like explicit state Model Checking, Action Planning, Game Solving and Probabilistic Model Checking proposing strategies to find solutions for the search problems.

Providing an universal search strategy which can be used in all environments to utilize solid state media and graphics processing units is not possible due to the heterogeneous aspects of the domains. Thus, this work presents a tool kit of strategies tied together in an universal three stage strategy. In the first stage the edges leaving a node are determined, in the second stage the algorithm follows the edges to generate nodes. The duplicate detection in stage three compares all newly generated nodes to existing once and avoids multiple expansions.

For each stage at least two strategies are proposed and decision hints are given to simplify the selection of the proper strategy. After describing the strategies the kit is evaluated in four domains explaining the choice for the strategy, evaluating its outcome and giving future clues on the topic.

Acknowledgments

In most cases a thesis would not exist without a Ph. D. Supervisor, but in this one the influence of Prof. Dr. Stefan Edelkamp, my supervisor, started much earlier. Being a diploma student he introduced me to the art of Model Checking. I do not know if it was because of *someone had to do the job* or because of *you are the right for the job* but he always motivated me. Prof. Dr. Edelkamp trusted me, much more then I could trust myself, and now you can read the results. Thanks for the endless discussions, on- and off-topic. Thanks for your time whenever I needed it. Thanks for the possibility to find oneself on the long line.

Special thanks go to Prof. Dr. Bernhard Steffen the man with the big picture. Although he never was the one I discussed implementation details with, he was always interested in my work and pushed me in the right direction when I stood at a forking way not knowing where to go. Thanks also for the warm place for my research.

I am also grateful to the other members of the committee: Prof. Dr. Jan Vahrenhold and Dr. Ingo Battenfeld for the help and support in the *last minutes*.

Thanks to all the coauthors who showed me the right way to write papers, thanks to Dragan Bosnacki, Pavel Šimeček, and especially to Shahid Jabbar. Pavel, perhaps one day we can play a second match *Czech Republic against Poland*?

When Stefan is my *Doctorvater* then Shahid certainly is my *Doktorbruder*. There are two images I see in front of me when thinking about him. I once came into his room and he was working on his thesis, he was adjusting a line in one picture, at the highest zoom level, moving it only some millimeters. It had to be perfect. The other image is a huge number of full and empty boxes in his room. I helped him to transport them to the local UPS store, the evening before his last flight to Sweden. Absolutely chaotic. This made him a human. Staying in the terms of a *Doktorfamilie* I will never forget my second *Doktorbruder* Peter Kissmann. Thanks for destroying my ideas at the right time. Peter is a gifted person in my eyes, his gift is to smell inconsistencies before his discussion partner has formulated the whole idea. It was not always funny to think a whole weekend on a particular algorithm and then see it collapse because of an overseen littleness. And of course thanks for the rigorous proof-reading of this work, I hope it makes it readable.

At this point it is time to thank the whole LS5 Team from the present and the past. I can not remember all the names due to a miserable memory, but i try to remember some. Thanks to the proof readers Julia Rehder, Falk Howar, Maik Merten and Johannes Neubauer. Thanks go to Thomas Wilk for showing me skills in table soccer I will never reach. Thanks to the, sporting ace Christian Wagner for motivation on this domain, and

Sven Jörges for the photo finish in submitting a dissertation.

All the time during this work, there was only one source of energy for me, this source was, is and will always be my family. I would like to return all of what you gave me, but I am sure I will never be able to. Names given in a text have to be in an order so I order them by age, but trust me Son, Wife, Mom and Dad I did all this for you, for you all.

Last but not least I would like to send special thanks to Cengizhan Yücel a long time student friend and the one of the most reliable persons i ever knew.

Finally a big thanks goes to the *Deutsche Forschungsgeselschaft* for financially supporting my research through the project *Modellprüfung auf Flashspeicher-Festplatte und Grafikkarte*.

Contents

III Action Planning 115

IV Game Solving 137

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Time is a scarce resource, space is unlimited. This statement is becoming reality in the 64-bit era. Today, an ordinary personal computer utilizes up to 32 gigabyte of internal RAM storage, using server hardware even 256 gigabyte in one computer are possible. Utilizing external storage one can get 3 terabyte magnetic drives, so called *Hard Disk Drives* (HDDs) at nearly 100 € and continuously falling prices. An example of resource capabilities these days is the company *Google*, who is offering an Email service giving each user 7.44 gigabyte available space for their mails. In February 2010 the service had reported to have 170 million accounts[1] with an available space of 1,235,156.25 terabytes or 1.177 exabytes.

For applications where excessive random access to the data is mandatory and magnetic drives fail to reach appropriate speeds, solid state drives (SSDs) have entered the market. An SSD stores information on memory chips providing faster access to the data while consuming less power. A system with 256 GB of RAM costs about 4,500 € at today's prices while the same amount of SSD storage costs nearly 300 € utilizing a reading access speed of 255 MB/s (compared to about 100 MB/s on HDDs and 17 GB/s for RAM) and the possibility to be extended by adding more or larger devices. The characteristics of solid state media differ significantly from the ones of hard disk devices imposing new challenges on the developers of algorithms. While writing data to such a medium is done at a speed comparable to magnetic devices (being about 100 MB/s) the reading of random bits can be done with much higher efficiency. These characteristics, additionally with the possibility to even increase the throughput by combining several devices, make them perfect for storing random access structures which exceed the size of RAM.

Rising storage capabilities do not necessarily require a change on the algorithmic level. In contrast to this, the switch from increasing the clock rate to assembling multiple cores in one central processing unit (CPU) demands for a parallelization on the algorithmic level. This change in design forces the algorithms to utilize the available

[1] http://news.bbc.co.uk/2/hi/8506148.stm

parallel computing power to gain profit from it and requires synchronization techniques and load balancing. The current maximum number of cores in one CPU is 8 which simulate 16 cores by utilizing two threads per core. Being confronted with a number of different tasks in parallel, the CPU cores are self-contained computing cores even able to increase the clock for one single core to speed up sequential computation. Simultaneously to the CPU the developers of the graphics cards increased the computation power by parallelization. Contrary to data processing a graphics processing unit (GPU) is used to compute the visualization for a large number of triangles representing a virtual world. Since the triangles are independent and the computation is equivalent for all of them, the parallelization used is the single instruction multiple data (SIMD) technique. Here a large number of processors manipulate data using the same instructions. Current GPUs utilize up to 1,024 processors in one graphics card (NVIDIA GTX 590) and up to 4 cards can be combined in one system.

This work utilizes recent developments in hardware to solve search problems in which the goal is to find a set of explicit nodes in a graph defined implicitly prior to the search. In an *implicit* definition just the starting node and a *transition function* transforming this node into new ones is given. The main challenge spreading over all implicit graph search problems is the *state space explosion problem*, which describes the potentially exponential development of node numbers in the graph. Even small changes in the transition function definition may increase the number of *reachable* nodes, on a path from the initial node, dramatically increasing the search time, which is mostly linear to this number.

Between all the available search challenges given in the scientific and non-scientific work, the four investigated domains represent a spectrum and give a starting point for investigation in many other research areas.

The first investigation on the usage of solid state drives and graphics processing units is *Explicit State Model Checking* (Clarke *et al.*, 1999; Müller-Olm *et al.*, 1999), where the demand for storage space and computation power increased dramatically with the introduction of parallel processors. It should be needless to say how important software verification has become in the last years. With the introduction of concurrent hardware at affordable prices for everyone, parallel programming has evolved to a standard technique to implement efficient algorithms. Not so long ago, only security related bugs were hunted, or bugs whose removal would directly avoid loosing equipment worth millions of dollars like an exploding space rocket. Today all companies search for efficient ways to verify their software because a bug can cause a significant loss of reputation resulting in the emigration of customers. The automobile constructor Mercedes-Benz learned their lesson when long term clients switched to other manufacturers because of small bugs in the car software. Even though the bugs were not dramatic, e. g., a not opening door when the remote was pressed, the damage to their reputation of being a premium manufacturer was worth millions of dollars due to decreasing sales.

The second chosen discipline is *Action Planning* (Russell and Norvig, 2002), where the goal is to find a plan fulfilling predefined conditions given a set of actions. The plan consists of a sequence of transition functions (here denoted as *actions*) which transform the initial state into a goal state. Prominent examples of planning are logistic domains as well as planning robots which perform various tasks efficiently. This work deals with deterministic Action Planning where each action is fully defined.

A breadth-first state-space generation in the artificial intelligence (AI) branch *Game Solving* (van den Herik *et al.*, 2002) imposes new challenges on the SSD and GPU utilization. Problems in this domain are usually built up of a high number of available moves, e. g., all possible movements to the figures on a chess board, with only a small number of these being valid moves. While a check for a single move can be done very efficiently the high number of checks imposes a long searching time. An additional aspect of this problems is a large state description where an efficient compression is necessary to traverse the entire state space. Here the computation power of the GPU comes in handy since the decompression and the determination of valid movements can be done in parallel on a huge number of states.

Probabilistic Model Checking (Kwiatkowska *et al.*, 2007) has been proved to be a powerful framework for modeling various systems ranging from randomized algorithms via performance analysis to biological networks. Although solutions for Probabilistic Model Checking can also be obtained by a state space search for a specific state, this is not an efficient way. Here the satisfaction of properties is quantified with some probability in contrast to the previous disciplines. In a state space approach this maps to generating states and annotating them with a probability until the target is reached. Due to the high branching factor it is more efficient to choose a different approach i. e., using numerical methods which enforce different strategies to utilize the GPU and external media. In this discipline the GPU has to be used with the intention of solving linear equations, imposing new challenges on the algorithm development.

This work will investigate in using recent developments in hardware to allow for traversing larger graphs in less time in all these domains.

1.2 State Space Exploration

The connecting aspect of all analyzed problems is the traversal of a search graph defined only by a starting node and a transformation function. To understand the correlation exploited in this work we need to define the basics of the *state space exploration* and present a number of existing algorithms. The following sections will provide the necessary definitions to explore state spaces and to analyze the proposed algorithms.

1.2.1 Introducing State Spaces

As this dissertation concentrates on *implicitly* given graphs where only a starting point and instructions how to traverse the graph are given we limit the definitions to those graph structures.

Definition 1 (System) *A* system *is a problem definition in a given environment. It includes all the necessary information to solve the problem and can be expressed in a suitable description language.*

A system usually consists of three aspects, a description of an environment, definitions of transformable elements in this environment and transition functions for these elements. Board games are systems where the board defines the environment and the pieces are placed or moved by transforming their position. The rulebook defines the transition functions by describing allowed moves.

Definition 2 (State) *A* state s *is a description representing the overall configuration of a system at a specified point in time.*

A state can be the concatenation of sub-states each describing a part of the system. To give an example take a look at the game checkers, here each piece can be a usual *men* or a *king*. One solution to store this difference in a state is to use different notations for this attribute. Another solution is to include this as a special variable representing this piece in the state. This variable is called *local state* since a change to this local state is only locally in the whole system.

Definition 3 (Local state) *The* local state *of an unique actor in a system is a variable describing the current condition of the actor.*

As an example, the local state of a piece denotes whether it is a men or a king and a state is the position of all pieces on the board in checkers. Replacing or removing pieces and changing a local state when reaching the appropriate position corresponds to transforming one state into another.

Definition 4 (State Space) *A* state space $\mathcal{S}$*, is the set of all possible configurations of a given system.*

The state space $\mathcal{S}$ of the game chess consists of all the possible placements for all pieces. $sin\mathcal{S}$ is mapped to a node $v \in V$ in the graph $G = (V, E)$.

A subset of all states $\hat{s} \in \mathcal{S}$ identifies the *initial* states, defined entirely before the search. All systems analyzed in this work only have one single initial state. The set of *reachable* nodes is a subset $r \subseteq \mathcal{S}$ denoting all nodes connected from $\hat{s}$.

To define the remaining states the informal definition of the transition function is formalized and *transitions*, composed of a *precondition* and a *postcondition* are defined as follows,

Definition 5 (Precondition) *A* precondition *of a transition defines conditions in the state to be true before transforming the state by applying a transition.*

Definition 6 (Transition) *A* transition t *in the set of all transitions* T *is a pair connecting one Boolean precondition and a set of postconditions. The transition is called* active *when the precondition evaluates to true.*

A transition has to define the modifications to the state in a set of postconditions.

Definition 7 (Postcondition) *The* postconditions *define conditions in the state which have to be true after it has been transformed.*

A transition from $s_1 \in \mathcal{S}$ to $s_2 \in \mathcal{S}$ is similar to an edge in the directed graph $G = (V, E)$ with $V = \mathcal{S}$ and $E = (v_1, v_2)$ where $v_1 = s_1 \wedge v_2 = s_2$.

Both, the precondition and all postconditions may be empty, resulting in a transition applicable to all states or a transition transforming the state into its identity. While the identifier precondition is common in Action Planning the postconditions in Action Planning and Model Checking are called *effects*. The later additionally utilizes the term *guard* to denote a precondition. In Game Solving the preconditions are given as rules whether a move is valid or not, the postcondition is the result of executing a valid move. Probabilistic Model Checking is an exception where all preconditions are active with a given probability. Postconditions are applied when a precondition is chosen.

Definition 8 (Parents and Successors) *When applying a transition the base state is called* parent *while the resulting state is called* successor. Leaves *are states without successors having no active transition.*

Definition 9 (Expansion / Generation) *During the* expansion *of a parent, or the* generation *of successors, the parent is* expanded *when all successors have been* generated*.*

While definitions so far applied to single nodes in a graph, the following one will cover connected nodes.

Definition 10 (Path) *A* path *is a sequence of states $s_0, s_1, \ldots, s_n$ where an active transition exists for all pairs (s_i, s_{i+1}) with $0 \leq i < n$. The* length *of a path is n the number of states it contains.*

Starting at an initial state and transforming it into a number of successors will generate a tree. To transform this tree into a graph *duplicates* have to be defined, describing states which are indistinguishable.

Definition 11 (Duplicate) *Two states s_1 and s_2 reachable on different paths from the initial state ($\hat{s} \ldots s_1 \neq \hat{s} \ldots s_2$) are* duplicates *when their representations are identical (write $s_1 = s_2$).*

Lemma 1 *In two state spaces $\mathcal{S}_1$ and $\mathcal{S}_2$ using the same set of transitions T and duplicate initial states $\hat{s}_1 = \hat{s}_2$, for each state $s' \in \mathcal{S}_1$ exists a duplicate state $s'' \in \mathcal{S}_2$ so that $s' = s''$ and $\mathcal{S}_1 = \mathcal{S}_2$.*

Proof. Since the representation of the duplicates $\hat{s}_1 \in \mathcal{S}_1$ and $\hat{s}_2 \in \mathcal{S}_2$ are identical the set of active transitions $a \subseteq T$ (the set of non-active transitions $\bar{a} : T/a$) is identical.

Applying the same postconditions of an active transition $a_1 \in a$ to $\hat{s}_1$ or $\hat{s}_2$ results in a new duplicate successor s for every $a' \in a$. The same applies to every s further down the path. □

Finally, after having introduced paths and duplicates the definition of a cycle can be given.

Definition 12 (Cycle) *A* cycle *in a state space is a path of arbitrary length connecting two duplicates.*

The shortest cycle is a path (s,s) formed by a transition without effects.

1.2.2 Example of a State Space

This section introduces an example to sketch a state and the state space on a constructed problem.

The problem is to finish a work denoted as *thesis* by a given actor called *student*. The student is supported by a variable number of *friends* to review the thesis and reduce its level of completeness by a given amount, due to pointing out errors and inconsistencies. The student alternates between *thinking* and *writing* of the thesis to complete it, but also has the necessity to *sleep* and *eat* during this process, while his friends are *enjoying time* or *correcting* the work.

To simplify the students life we set up some assumptions:

- After sleeping the student has to eat.
- Having eaten the student starts thinking on the thesis.
- The student immediately writes down her or his thoughts.
- If the student is neither hungry nor sleepy having finished writing a part he starts to think about further parts.
- Writing makes hungry.
- Eating makes sleepy.
- A friend can only review a thesis if something is written.

The question here could be if the work will be finished or how the number of reviewers influences the time to finish the work or to find a plan to distribute the thesis among friends efficiently.

Figure 1.1 visualizes the student's and one friend's behavior as a directed graph. Each circle is a local state denoted with a name in the upper half. Edges represent transitions from the parent state to its successor. If a precondition for a transition exists it is given the prefix *pre:* above the corresponding edge. Postconditions are described under the edges and prefixed with a *post:*. There is one global variable called completeness denoting the `completeness` of the work not visualized in the graph.

The state space of the student, depicted in Figure 1.2 shows a directed graph of all reachable states for the student, if no interaction with a friend appears. The name of the state is shown in the upper half, and the variables being `true` in a state are visualized in the bottom half of the circle. In contrast to Figure 1.1 states with similar names appear several times in the state space since the values of the variables in it differ. Figure 1.2 visualizes only the state space of the student omitting interventions of friends. A state space involving correction cycles is a cross product of the state spaces of all actors. So in each state the student is in the friend can be in an *enjoying time* or in a *correcting* state, increasing the number of states by a factor of two for each extra friend. Additionally, tracking the completeness variable in the state, e. g., as an integer value between 0 and 100% would theoretically blow up the state space by a factor of 100 making reduction and compression strategies essential.

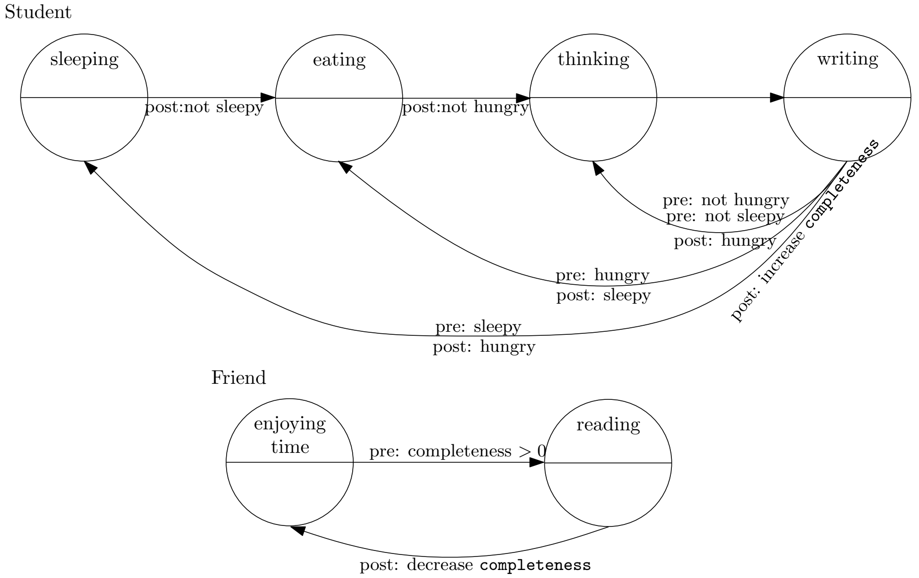


Figure 1.1: Visualization of the thesis problem as a graph. Circles denote a local state the actor can be in. Each local state is denoted with a name in the upper half of the circle. Edges represent transitions from the parent to the successor with preconditions above it prefixed by *pre:* and postconditions by *post:*. The upper graph presents the transitions for the student and the lower one those for a friend.

Both examples sketch the roots of the states space explosion problem in a simplified manner and motivate the necessity for efficient state storage and processing strategies. While a naive implementation of the `completeness` variable would increase the state space by a factor of 100 the binary representation can reduce the factor to $log(100) = 7$. Additional reduction techniques are abstraction (Edelkamp and Lluch-Lafuente, 2004; Namjoshi and Kurshan, 2000) and compression (Lluch-Lafuente *et al.*, 2002; Clarke *et al.*, 1994; Korf, 2008b; Holzmann and Puri, 1999).

1.2.3 State Spaces in the Following Parts

Although the following domains seem to be very different the strategy of state space exploration is the connecting aspect. Model checking, Action Planning, Game Solving and Probabilistic Model Checking are all search problems and easily mapped to a graph algorithm. The proposed technique to use recently developed hardware in graph searching is the roof standing on four pillars, depicted in the four disciplines. The next sections will sketch the mapping of each part to graph searching while a detailed mapping is given in each corresponding part.

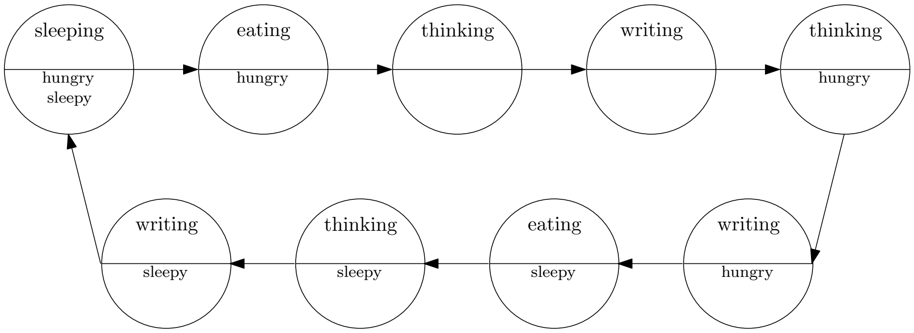


Figure 1.2: The state space containing 9 states the actor *student* defined in the thesis problem can be in. The communication with friends, who correct the thesis, is avoided due to complexity of the visualization. When an additional actor (e. g., friend) is included each state is extended by a local state of the actor increasing the number of states by a factor corresponding to the number of local states the actor can be in. This simplification also abstracts from the *completion* variable in the states which would increase the number of states by a factor of 100 if used as a percent variable.

Model Checking

In *Model Checking* a model, describing a system and given in a description language is checked for a given property. Here, the system is defined prior to the search, given variables and processes as transformable elements, followed by transition functions denoted as *transitions*. Although the environment is not given explicitly it is given by the model checker who is handling the description language. The goal of finding states violating the property is achieved by checking each single state reachable from an initial state against it. To check lifeness properties a so called *lasso* path has to be found. Such a path consists of a cycle containing at least one special (in this discipline denoted as *active*) state and that must be reachable from the initial state and defined in the model description. For such a search, special graph traversal algorithms were developed and Part II will propose an extended algorithm. It uses a number of standard graph search algorithms to find the shortest lasso in a state space. The following chapter proposes an approach to efficiently utilize graphics cards when generating the state space in Model Checking.

Action Planning

Action Planning is a scientific domain for finding a plan in a given environment to achieve a defined goal. The system of Action Planning is an environment for an actor, e. g., a robot. The actor has to perform actions to find a sequence of actions, called plan, to put himself, into a given goal configuration. This plan can be mapped to a path in a graph, starting at an initial state and connecting it to a state where the goal is achieved. The initial state is defined prior to the search and transitions are given by actions in a description language. The way to find such a plan is to generate all states until the goal state is reached and then either search backwards to the initial state, for a plan reconstruction, or store the plan while searching. Refined Action Planning uses *costs* which map each action to a value to generate a more realistic representation. Here, an exploration considering only the length of a path is not efficient and special graph algorithms (e. g., Dijkstra's Algorithm (Dijkstra, 1959)), are used. Part III presents a graphics card algorithm extended to support action costs and external media for a generation of the plan.

Game Solving

The system of a game in *Game Solving* is the state of the game at a specific stage. In board games it suffices to represent the board and the positions of all pieces on it in the system. The transitions are the rules of the game given prior to the search. The task to decide whether a given player can win the game at a given state is achieved by visiting every state and checking for a path to a winning state. One approach to solve a game is a two searches strategy. In the first *forward* search all states reachable from the initial state are generated and classified whether they are winning states for a player or not. The second *backwards* search starts at all winning states and propagates the information which player has won to the predecessors. In games with only one winning state, like one player combinatorial games, a forward search from the current state suffices to determine if the game can still be completed. In two player games all terminating winning states have to be identified by a forward search followed by a backward search from these to classify all states up to the initial state. Part IV proposes to compress each state to a number by using a permutation rank strategy or binomial and multinomial hashing to decompress the state on the graphics card and analyze it. This strategy can be evaluated efficiently due to the highly parallel processing power of this unit.

Probabilistic Model Checking

Probabilistic Model Checking avoids the preconditions of the state space search by replacing them with the probability of being active. The probabilities of all preconditions in one state sum up to 100%. On leafs this is achieved by adding an outgoing transition without postconditions having a probability of 100%. A naive graph theoretical approach to determine the probability of a property violation is to find a path from the initial state to a violating one and compute the probability along it. This approach can be very ineffective in terms of computation time and usually a different technique is used. The state space is mapped to a matrix with the probabilities given in that matrix and the probability is computed by solving a set of linear computations using a matrix-vector multiplication approach. Part V decreases the time to find a solution significantly by porting the solving process partially to the graphics card.

1.3 Graph Search Algorithms

Moving from node to node in a graph, respectively traversing a state space, requires a strategy, including a storage- and a decision-rule for the order the successors are generated in. This section will develop a basic algorithm and extend it to more sophisticated strategies optimizing it for different conditions like generating speed or search direction. The development starts with a *blind search*, not using information about the preferred transitions, and resorts to some form of cost-first shortest path path exploration, which requires costs assigned to the edges given in the graph description. Since state spaces can become arbitrarily large the section also introduces algorithms for *external search*, which utilizes external media like hard disks to store information, and *parallel search* utilizing parallel hardware.

1.3.1 Blind Search

In blind search the order of state expansions is defined by the search algorithm, information about preferred transitions is omitted. While generating a state in a search algorithm the generated successors have to be stored for a potential expansion in the further traversal in a dedicated structure called *open list*.

Definition 13 (Open list) *The set of generated, but unexpanded states is called an $Open$ list (or just* Open*) also denoted as a* working set*.*

Using only an $Open$ list one can already form an algorithm which is *complete*, so it will visit all states in the given state space provided it is circle free.

Algorithm 1.1: Graph algorithm using an $Open$ list

```
Input: $\hat{s} \in \mathcal{S}$ initial state, $T$ set of transitions
$Open \leftarrow \hat{s}$ ;                                   {store $\hat{s}$ in $Open$ }
while $Open \neq \emptyset$ do                               {repeat until search terminates}
    choose a state $s \in Open$ ;                           {usually the first one in the list}
    expand successors $s \to s_1 \dots s_\nu$ ;   {apply transitions to generate successors}
    for $s_i$ $(\forall i : 1 \leq i \leq \nu)$ do                       {check each successor}
        if $s_i \notin Open$ then                           {when not already in $Open$ }
            $Open \leftarrow s_i$ ;                                    {add it to $Open$ }
    remove $s$ from $Open$ ;      {all successors generated so state can be dropped}
```

After inserting the initial state into the $Open$ list, Algorithm 1.1 generates the successors of a state by checking the preconditions and applying corresponding postconditions and stores them in $Open$. When all successors of a state are generated and inserted into $Open$ the state is removed from the list.

Lemma 2 *Algorithm 1.1 will terminate and expand all paths in the state space, visiting all states, if the state space is cycle free.*

Proof. Each state remains in $Open$ until all its successors are generated. Removing a fully expanded state is safe since all paths crossing this state to its successors are extended by at least one state. Leaves, states without successors, are end points of paths which cannot be extended and are removed from $Open$ immediately when generated.

Since the state space is cycle free each path from $\hat{s}$ has to end with a leaf forcing the algorithm to terminate. □

For state spaces containing cycles Algorithm 1.1 has to be extended. Consider a transition set T with two transitions $\{(\hat{s}, s), (s, \hat{s})\}$ from the initial state to a successor and back to the initial state. The algorithm will add the initial state to $Open$ over and over again, being trapped in the cycle unable to terminate. To avoid this behavior an option is needed to decide whether a duplicate of a state was already removed from $Open$, thus the states removed from $Open$ are stored in a separate structure.

Definition 14 (Closed list) *When all successors of a state are generated it is moved to the* $Closed$ *list (usually just* Closed*), also denoted as* visited set*.*

Algorithm 1.2: Graph algorithm using an $Open$ and a $Closed$ list

```
   Input : $\hat{s} \in \mathcal{S}$ initial state, $T$ set of transitions
 1 $Open \leftarrow \hat{s}$ ;                                   {store $\hat{s}$ in $Open$ }
 2 $Closed \leftarrow \emptyset$ ;                               {clear $Closed$ list}
 3 while $Open \neq \emptyset$ do                          {repeat until search terminates}
 4     choose a state $s \in Open$ ;                       {usually the first one in the list}
 5     expand successors $s \rightarrow s_1 \dots s_\nu$ ; {apply transitions to generate successors}
 6     for $s_i$ $(\forall i : 1 \leq i \leq \nu)$ do                   {check each successor}
 7         if $s_i \notin Open \wedge s_i \notin Closed$ then        {when not already expanded}
 8             $Open \leftarrow s_i$ ;                                {add it to $Open$ }
 9     remove $s$ from $Open$ ;  {all successors generated, so state can be removed}
10     $Closed \leftarrow s$ ;                                    {add $s$ to $Closed$ }
```

Algorithm 1.2, which extends Algorithm 1.1 by a $Closed$ list, detects duplicates using the lines 6 to 8 and avoids adding them to the $Open$ list.

Lemma 3 *Algorithm 1.2 will terminate and expand all states reachable in the state space, visiting each state exactly once.*

Proof. For state spaces without cycles the duplicate detection is not needed, here the proof of Algorithm 1.1 can be applied.

Let us assume a cycle exists and state s_c is the first reached state on this cycle. Line 8 ensures that s_c is stored in $Open$ on the first appearance and line 10 stores it in $Closed$ when it has been expanded. While extending all paths crossing s_c the algorithm will reach it again, but avoid inserting it into $Open$ since it was already inserted or expanded. When a duplicate of s_c exists it will also be bypassed which does not matter since the states behind this duplicate also exist behind s_c. □

In the example given above, with transitions $\{\hat{s}, s), (s, \hat{s})\}$, Algorithm 1.2 will not add $\hat{s}$ a second time to $Open$ since it is already present in $Closed$. Although the pseudocode is extended only in two lines the problem of looking up a state in the $Closed$

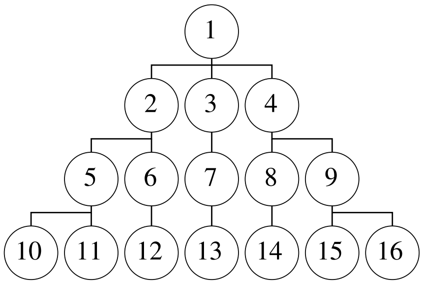


Figure 1.3: BFS ordering of states.

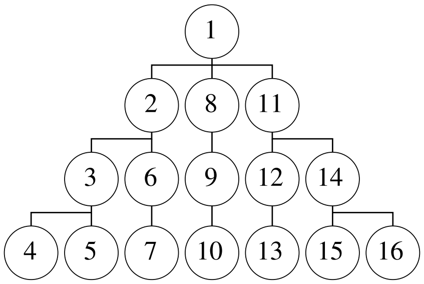


Figure 1.4: DFS ordering of states.

list should not be underestimated. Many strategies exist and scientists are still developing new ways to either perform or avoid a random lookup, or store $Closed$ efficiently on external media without the necessity to perform a scan through the complete file for each generated state. Based on Algorithm 1.2 several strategies were developed to traverse state spaces efficiently, and this work is another contribution to these strategies.

The most prominent algorithms are *Breadth-First search* (BFS) and *Depth-First search* (DFS) (Knuth, 1973). The difference between those algorithms is only the order of storing states in $Open$. BFS stores them in a First In / First Out strategy while DFS uses a Last In / First Out $Open$ structure.

Algorithm 1.3: Breadth-First search

```
Input  : $\hat{s} \in \mathcal{S}$ initial state, $T$ set of transitions
$Open \leftarrow \hat{s}$ ;                                   {store $\hat{s}$ in $Open$ }
$Closed \leftarrow \emptyset$ ;                               {clear $Closed$ list}
while $Open \neq \emptyset$ do                                {repeat until search terminates}
    choose first state $s \in Open$ ;
    expand successors $s \rightarrow s_1 \dots s_\nu$ ; {apply transitions to generate successors}
    forall the $s_i$ $(\forall i : 1 \leq i \leq \nu)$ do      {check each successor}
        if $s_i \notin Open \wedge s_i \notin Closed$ then     {when not already expanded}
            $Open \leftarrow s_i$ ;                            {add it to the end of $Open$ }
    remove $s$ from $Open$ ;   {all successors generated so state can be removed}
    $Closed \leftarrow s$ ;                                    {add $s$ to $Closed$ }
```

Algorithm 1.3 visits all states ordered by the distance to $\hat{s}$ while Algorithm 1.4 visits states with a maximal distance to $\hat{s}$ first. In contrast to Algorithm 1.2 the order of storing states in the $Open$ list is given explicitly by the algorithm.

Since only the order of storing the states in $Open$ is different to the general algorithm the proof of completeness is inherited from the previous algorithms. The difference in the order of visiting nodes is displayed in Figures 1.3 and 1.4.

Algorithm 1.3 partitions $\mathcal{S}$ into *BFS-Layers*. All states in a BFS-Layer have the same distance from the initial state.

Algorithm 1.4: Depth-First search

```
Input : $\hat{s} \in \mathcal{S}$ initial state, $T$ set of transitions
$Open \leftarrow \hat{s}$ ;                                          {store $\hat{s}$ in $Open$ }
$Closed \leftarrow \emptyset$ ;                                      {clear $Closed$ list}
while $Open \neq \emptyset$ do                                       {repeat until search terminates}
    choose first state $s \in Open$ ;
    expand successors $s \rightarrow s_1 \ldots s_\nu$ ;             {apply transitions to generate successors}
    forall the $s_i$ $(\forall i : 1 \leq i \leq \nu)$ do            {check each successor}
        if $s_i \notin Open \wedge s_i \notin Closed$ then           {when not already expanded}
            $Open \leftarrow s_i$ ;                                  {add it to the beginning of $Open$ }
    remove $s$ from $Open$ ;                 {all successors generated so state can be removed}
    $Closed \leftarrow s$ ;                                          {add $s$ to $Closed$ }
```

Table 1.1: Main differences between BFS and DFS.

	BFS	DFS
speed	slow	fast
Open size	bound by largest layer	bound by depth
cycle detection	none	by checking new states in *Open*
distance to initial	minimal	not specified

Although the difference in pseudocode is marginal the impact on the evaluation of the algorithm is not. Table 1.1 points out some of the main differences. The DFS algorithm turns out to be much faster on today's hardware due to its better cache efficiency. Although the work to expand all states is the same the BFS algorithm stores a large number of states in memory and fetches a state from a distant region of it for expansion. In contrast DFS expands the last generated state which resides often still in the cache of the CPU. On the other hand the BFS algorithm can be parallelized trivially by sending generated successors to different nodes. For the DFS algorithm an efficient parallelization is much harder to realize since only one successor of a parent is generated. Memory consumption of *Open* also differs significantly in both approaches, while in BFS the next BFS-Layer has to be stored in *Open* DFS stores the path from the initial state to the current one. This path is especially short in state spaces with a low BFS-depth but a high branching factor. Storing the entire path from initial also enables a trivial cycle detection extension to the algorithm. By simply checking each generated state for a duplicate in *Open* all cycles reachable from $\hat{s}$ are found. In BFS this strategy fails due to all states in *Open* having the same distance to the initial. However, in BFS this distance is guaranteed to be minimal while in DFS the depth at which a state is found depends highly on the state space and the chosen successor to generate.

To connect the optimality in depth and the speed of single expansions Korf (1985) presented *iterative deepening* (Korf, 1985) as described in Algorithm 1.5. Here a DFS is started with a maximal depth max_d given before the search. When the desired goal

Algorithm 1.5: Iterated Depth-First search

```
Input  : $\hat{s} \in \mathcal{S}$ initial state, $T$ set of transitions
Output: minimal path to goal state if it exists
$Open \leftarrow \hat{s}$ ;                                   {store $\hat{s}$ in $Open$ }
$Closed \leftarrow \emptyset$ ;                               {clear $Closed$ list}
$max_d \leftarrow 2$ ;                                        {depth bound for first iteration}
while $iterate$ do                                            {repeat until whole state space generated}
    $iterate \leftarrow$ false ;                              {variable to force another iteration}
    while $Open \neq \emptyset$ do                            {repeat until search terminates}
        choose first state $s \in Open$ ;
        expand successors $s \rightarrow s_1 \ldots s_\nu$ ;
        {apply transitions to generate successors}
        forall the $s_i$ $(\forall i : 1 \leq i \leq \nu)$ do  {check each successor}
            if $s_i \notin Open \wedge s_i \notin Closed$ then   {when not already expanded}
                if $|Open| + 1 > max_d$ then                  {states exist under the bound}
                    $iterate \leftarrow$ true
                else
                    $Open \leftarrow s_i$ ;                   {add $s_i$ to the beginning of $Open$ }
                if $s_i \in goal$ then return $Open$;         {return path to goal}
        remove $s$ from $Open$ ;
        {all successors generated so state can be removed}
        $Closed \leftarrow s$ ;                               {add $s$ to $Closed$ }
    $max_d \leftarrow max_d + 1$ ;                            {increase depth bound}
```

state is not found the depth bound is increased and the search restarted. When no length of a path exceeds the bound the search terminates. This algorithm is not complete, it does not necessarily expand all states up to the given search border.[2] Although a DFS is used, the path delivered to the goal is minimal due to the increasing border by one and a goal is reported at the minimal depth bound.

All blind algorithms assume that transitions are preferred according to the given expansion strategy. When the state space description includes an ordering on the transitions the algorithm has to take this into account while expanding. One possibility used in planning to impose an ordering on transitions is assigning them *costs* of evaluation by defining a *cost function*.

Definition 15 (Cost Function) *A* cost function *cost is a mapping $T \rightarrow \mathbb{R}$ assigning each t $\in T$ a cost value.*

[2] Assume a search border of b for a given iteration and a state s in a depth b then s can be reached by the search and stored in *Closed*. When reached again in a lower depth it will not be expanded due to its existence in *Closed* and its successors will be omitted.

Analogously the cost of a path is defined as follows.

Definition 16 (Cost of a Path) *The* cost *$cost\ (s_0 \ldots s_n)$ for a given path $(s_0 \ldots s_n)$ is the sum of all costs of transitions applied in the path*

$$\sum_{i=0}^{n-1} cost(t(s_i, s_{i+1}))$$

.

In state spaces with uniform costs, e. g., $cost(t) = c\ \forall t \in T$, the length of a path conforms to $cost(s_0, \ldots, s_{n-1})/c$.

Given a cost function, and interested in the path with minimal costs, Dijkstra presented a graph traversal algorithm in 1959 expanding nodes in the order of increasing costs. Algorithm 1.6 applicable to state spaces stores pairs $(s, cost(s))$ in $Open$ ordered by $cost()$ to compute the summarized costs for a path.

Algorithm 1.6: Dijkstra's Algorithm

```
Input: $\hat{s} \in \mathcal{S}$ initial state, $T$ set of transitions, $cost : T \rightarrow \mathbb{R}$ transitions to costs
       mapping
Output: $pathcost$: minimal costs for each state reached on a path from $\hat{s}$
$Open \leftarrow (\hat{s}, 0)$ ;                                   {store $\hat{s}$ and $cost\ (\hat{s})$ in $Open$ }
$Closed \leftarrow \emptyset$ ;                                          {clear $Closed$ list}
while $Open \neq \emptyset$ do                               {repeat until search terminates}
    choose the state $s \in Open$ with minimal costs ;   {e. g., in a priority queue}
    expand successors $s \rightarrow s_1 \ldots s_\nu$ ; {apply transitions to generate successors}
    forall the $s_i$ $(\forall i : 1 \leq i \leq \nu)$ do                    {check each successor}
        if $s_i \notin Closed$ then                      {look into $Open$ AND $Closed$ }
            $Open \leftarrow (s_i, cost(s_i))$ ;          {store $state_i$ and $cost(s_i)$ in $Open$ }
        else
            if $s_i \in Open$ and $cost(s_i) < cost(duplicate\ in\ Open)$ then
                $cost$(duplicate in $Open$) $\leftarrow cost(s_i)$ ;
                {update the cost in $Open$ }
    remove $s$ from $Open$ ;   {all successors generated so state can be removed}
    $pathcost \leftarrow (s, cost(s))$ ;                       {store $s$ and $cost(s)$ to return it}
    $Closed \leftarrow s$ ;                                              {add $s$ to $Closed$ }
return $pathcost$ ;
```

Algorithm 1.6 looks in $Closed$ for duplicates but also checks for existence of the state in $Open$ which is mandatory to update the costs for already generated but still not expanded states which were reached again using a cheaper path. To maintain $Open$ sorted priority queues (Edelkamp and Wegener, 2000; Cormen *et al.*, 2001) are used to speed up the algorithm.

Table 1.2: Dijkstra ordering of nodes in $Closed$. The position is given in the upper array and the path cost in the lower. The nodes are expanded in the order of their path costs.

Position	1	2	3	4	5	6	7	8	9	...	13	14	15	16
Path costs	0	1	2	3	3	3	4	4	5	...	5	6	7	8

Lemma 4 *Algorithm 1.6 returns the minimal cost of a path for all states reachable from the initial one for state spaces with non-negative costs.*

Proof. The algorithm partitions the state space in layers of equal costs. First consider state spaces with uniform costs $cost(t) = c$ $(\forall t \in T)$. Here the algorithm partitions the state space according to the BFS-Layers since the path costs start with 0 at the initial state and increase by c with every added state. $Open$ is always strictly sorted by the costs and every new state is added at the end having costs equal or higher to the previous one

- if a state s_i is a successor of s its costs are $cost(\hat{s}, s_i) = cost(\hat{s}, s) + c$.
- if two states s_i and s_j are successors of a state s, the costs are $cost(\hat{s}, s_i) = cost(\hat{s}, s_j) = cost(\hat{s}, s) + c$.

In state spaces with non-uniform and non-negative costs $cost(t) = c \geq 0$ $(\forall t \in T)$ the state space is partitioned in layers of equal costs. Assume we sort $Open$ after each insertion. With non-negative costs we have $cost(\hat{s}, s_i \in Succ(s)) \geq cost(\hat{s}, s)$, and when a state $s_o \in Open$ is being expanded the costs to reach it $cost(\hat{s}, s_o)$ is minimal compared to all remaining states in $Open$. So each new generated state $s_i \in Succ(s)$ is sorted behind s and an update of its cost can move it only further to the front of $Open$ but not before s. When s is expanded its costs will never again be updated since for all states $s_o \in Open$ we have $cost(\hat{s}, s) \leq cost(\hat{s}, s_o)$ so all remaining states will be added to $Closed$, and to $pathcost$ in a non-decreasing order of costs. □

After adding a cost function to the state space given in Figures 1.3 and 1.4 with costs in the range of $1 \ldots 3$ the Dijkstra algorithm is applied, resulting in a state ordering given in Figure 1.5. The states contain the expansion order at the top and the path cost at the bottom of a state depicted by a circle. Table 1.2 depicts the numbers again to visualize the implied ordering by path cost.

1.3.2 External Search

State space traversal algorithms consume a huge amount of memory. Storing all states of an implicitly given graph in RAM can be impossible given on the size of the graph. Solving the problem by using compression is a solution, but even with compression a minimal size for a state exists limiting the amount of states which can be stored.

Another approach is to store the nodes on external memory e. g., a HDD. Since hard disk drive has different access properties then internal memory, new algorithms had to be developed to utilize it efficiently.

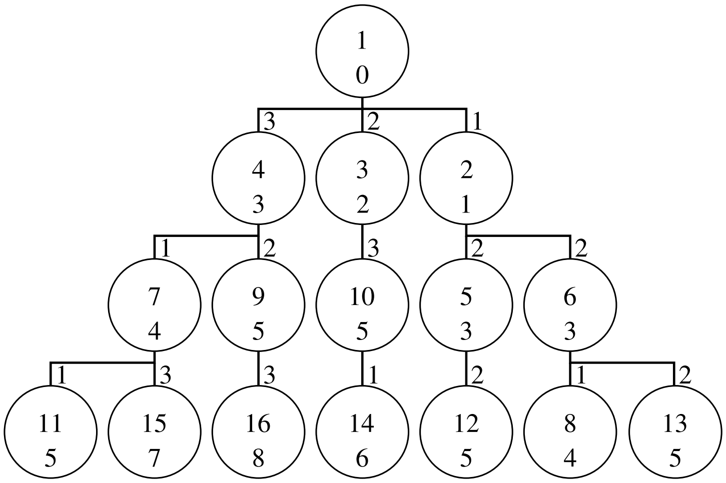


Figure 1.5: Dijkstra ordering of states in a virtual state space. The expansion order is given in the upper half, the path cost in the lower half of each circle. Edges are marked with the edge costs.

Data is stored in blocks on external media imposing a lag when accessing a random block or a single element in this block. Adjacent blocks of memory can be accessed without a latency, so accessing data is done sequentially since this strategy distributes the latency on all retrieved elements. Aggarwal and Vitter invented an adapted memory model to analyze algorithms that utilize external memory in 1988. Here accessing the data in blocks is preferred to analyze the performance of external memory algorithms. Graph traversal algorithms optimized for external memory usage are presented in (Meyer *et al.*, 2003).

Algorithm 1.7: The External BFS algorithm

```
Input : $\hat{s} \in \mathcal{S}$ initial state, $T$ set of transitions, buffer preferably size of RAM
$Open \leftarrow \hat{s}$ ;                                        {store $\hat{s}$ in $Open$ }
while $Open$ not empty do
    read $Open$ in buffer ;                                    {partially if to large}
    expand all states in buffer into a new file $Open_{next}$ on external device ;
    sort $Open_{next}$ ;                                       {externally if necessary}
    scan through $Open_{next}$ and remove adjacent duplicates ;
    scan trough $Open$ and $Open_{next}$ to remove duplicates from previous layers ;
    append $Open_{next}$ to $Open$;
```

Algorithm 1.7 presents a Breadth-First search approach (Munagala and Ranade, 1999), where all data is stored on external media. Internal memory is used as a *buffer* where nodes are stored temporarily before being written to the block device. Enabling an efficient duplicate detection is realized by sorting. Since the $Open_{next}$ file, containing all states to be expanded in the next BFS-Layer can potentially exceed the available buffer size an external sorting approach is needed. Being sorted, all duplicates in the file will be arranged adjacent to each other and can be removed easily by a single scan through the file. Mehlhorn and Meyer (2002) and later on Ajwani *et al.* (2007) present modifications of this algorithms with a reduced number of I/O operations due to caching an adjacency matrix of the nodes in the internal memory, reducing the running time by several folds.

While the presented modifications request an explicitly given graphs Korf (2003a) presents an implicit graph algorithm based on the idea of a *Frontier Search* analyzed in detail also by Korf in 2005.

Since the efficiency of an external algorithm highly depends on its implementation two C++ libraries exist to support the developer. The *standard template library for XXL data sets* (STXXL) (Dementiev *et al.*, 2005), used in the scope of this work, is the first I/O-efficient algorithm library that supports the pipelining technique. While the goal of the *Templated Parallel I/O Environment* (TPIE) is to provide a portable, extensible, flexible, and easy to use C++ programming environment.

1.3.3 Parallel Graph Search

Parallel graph search (an overview is given e. g., by Ghosh (1993)) increases the searching speed by utilizing more then one computation device to generate states or to check for duplicates and extends the available internal memory by using a distributed system. The challenge in parallel search is to find distinct parts of a graph and to distribute them to the computation cores.

Parallel search is divided in two sub domains called *distributed search* and *shared memory search*. Distributed search denotes the utilization of clusters, build up of a number of distinct computing systems connected through a network. Shared memory search relays on the existence of a memory accessible directly from all used computation devices.

Distributing a state space is usually done in one of two ways, either a static hash like function is used to determine which core is responsible for the state or a dynamic function analyzes the load on the cores and assigns a generated successor to a core with the minimal load. Both strategies are divided into further solutions to optimize the distribution and minimize the necessary communication.

A naive approach to parallelize the Breadth-First search (Ghosh and Bhattacharjee, 1984), depicted in Algorithm 1.8 is to distribute the generated successors among the available computing nodes. The problem with this realization is a high communication overhead between the nodes. Each generated state is send over a communication protocol to a distant node and a common $Closed$ structure has to be maintained to avoid the expansion of duplicates. Additionally to a $Closed$ synchronization the $Open$ structure of this algorithm has to be maintained on a *root* node.

The modified Algorithm 1.9 utilizes a hash function $h(s)$ to distribute states to indexed nodes enabling a distribution of $Closed$ and $Open$ structure. This *hash based partitioning* avoids the communication to determine already visited states but depends on the distribution of the hash function to achieve an efficient parallelization.

Algorithm 1.8: Basic parallel search strategy

```
Input : $\hat{s} \in \mathcal{S}$ initial state, $T$ set of transitions
$Open \leftarrow \hat{s}$ ;                                          {store $\hat{s}$ in $Open$ }
$Closed \leftarrow \emptyset$ ;                                      {clear $Closed$ list}
while $Open \neq \emptyset$ do                        {repeat until search terminates}
    choose n states $s_p \in Open$ ;
    forall the $s_p$ $(\forall p : 0 \leq p \leq n-1)$ do in parallel
        expand successors $s_p \rightarrow s_1 \ldots s_\nu$ ;
        {apply transitions to generate successors}
        forall the $s_i$ $(\forall i : 1 \leq i \leq \nu)$ do            {check each successor}
            if $s_i \notin Closed$ then                      {when not already expanded}
                $Open \leftarrow s_i$ ;                                {add it to $Open$ }
        remove all $s_p$ from $Open$ ;
        {all successors generated so states can be removed}
        $Closed \leftarrow s$ ;                                      {add $s$ to $Closed$ }
```

Algorithm 1.9: Hash based parallel search strategy

```
Input : $\hat{s} \in \mathcal{S}$ initial state, $T$ set of transitions, $N$ number of nodes
$Open[0] \leftarrow \hat{s}$ ;                               {store $\hat{s}$ in $Open$ of one node}
forall the nodes $n$ $(0 \leq n < N)$ do in parallel                      {start all nodes}
    $Closed[n] \leftarrow \emptyset$ ;                                  {clear local $Closed$ list}
    while $Open[0, \ldots, N-1] \neq \emptyset$ do              {repeat until all $Open$ empty}
        choose one state $s \in Open[n]$ ;
        if $s \notin Closed[n]$ then                          {when not already expanded}
            expand successors $s \rightarrow s_1 \ldots s_\nu$ ;
            {apply transitions to generate successors}
            forall the $s_i$ $(\forall i : 1 \leq i \leq \nu)$ do              {check each successor}
                if $h(s_i) \neq n$ then                {find responsible node for this state}
                    send $s_i$ to node $h(s_i)$ ;                {send it to appropriate state}
                else
                    $Open[n] \leftarrow s_i$ ;                             {add it to local $Open$ }
            remove $s$ from $Open[n]$ ;
            {all successors generated so state can be removed}
            store $s$ in $Closed[n]$ ;                              {add it local $Closed$ }
```

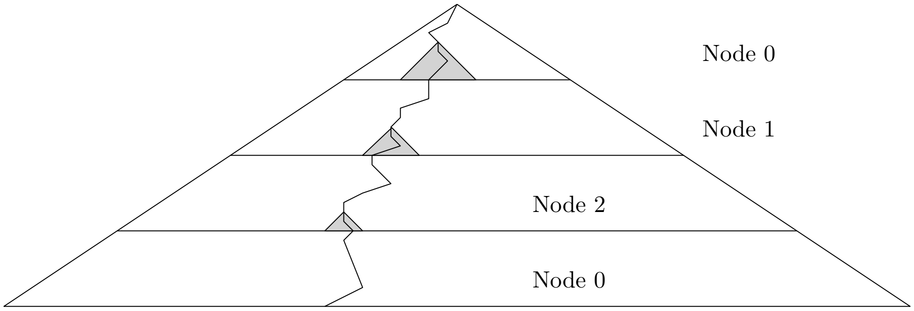


Figure 1.6: Distribution of nodes according to the DFS distance to the initial like proposed by (Holzmann and Bosnacki, 2007).

Zhou and Hansen (2007) presented a structured based parallel distributing approach which analyzes the state space prior to the search and distributes the states based on a state space abstraction function. Here, the advantage is the preferred expansion of successors on the same node, so each node can work on a specific region in the state space and distribute only distant states. They extended this strategy to a dynamic distribution technique. Here the state space is analyzed on the fly while searching and the distribution function adjusted (Zhou and Hansen, 2011). An adjustment includes a stop of the search and a rearrangement of already stored states.

While the previous parallelizations are based on BFS and can be used on shared memory and distributed systems, (Holzmann and Bosnacki, 2007) went a different approach and parallelized the Depth-First search on a shared memory multi-core system. Here, the states are distributed among the available cores depending on their DFS distance from the initial state like sketched in Figure 1.6. When all nodes are occupied the search continues on the first node which preferably expands states at a higher depth.

Recently Barnat *et al.* (2011) show how existing parallel algorithms to find strongly connected components in a graph can be reformulated in order to be accelerated by NVIDIA CUDA technology. In particular, they design a new CUDA-aware procedure for pivot selection and adapt selected parallel algorithms for CUDA accelerated computation.

DisNet, a tool set for Distributed Graph Computation (Lichtenwalter and Chawla, 2011) should be given as the latest example for a distributed graph search implementation. After supplying two small fragments of code describing the fundamental kernel of the computation. The framework automatically divides and distributes the workload and manages completion using an arbitrary number of heterogeneous computational resources.

Describing all possible ways to parallelize a state space search is sincerely out of the scope of this introduction and even not possible due to the large number. The sketch, given in this section is mentioned as a starting point for the following assumptions and development leading to an efficient algorithm for novel hardware.

1.4 Duplicate Detection in Graph Search

All state space searching algorithms rely on an efficient duplicate detection. In fact studies made within the scope of this work revealed the duplicate detection to consume over 50% of the whole searching time. So several strategies were developed to avoid re-expanding nodes. Removing already existing nodes requires a comparison function for state representations. This function can either compare the complete stored vectors or an approximated representation of them giving the chance to reduce the space needed to represent the state.

The challenge in duplicate detection is to find an already expanded state which is similar to the examined one, done either by sorting the new state into all existing ones or by looking up an entry in a table containing expanded states. While the sorting method is superior on block access devices with a slow random access, looking up an entry table is superior in memory structures with a short random access speed. Variations exist to speed up the checking process by either reducing the number of comparisons or the amount of memory used for the expanded states.

In special cases, where enough information of the state space is given prior to the search it is possible to reduce the duplicate detection to only special states or even avoid it completely. In his thesis Jabbar (2008) has shown that when exploring undirected graphs with the BFS algorithm a check of the previous two layers suffices to remove all duplicates.

External search introduces the term *Delayed Duplicate Detection* (DDD) in contrast to *Immediate Duplicate Detection* (IDD) defined as follows.

Definition 17 (Delayed/Immediate Duplicate Detection) *In* Delayed Duplicate Detection *(Korf, 2003a) the detection of duplicates is postponed to a specific point in the search, e. g., when one BFS-Layer is generated, to increase per state performance. Taking into account that states may be stored more then once.*

Immediate Duplicate Detection *checks for existing duplicates immediately after a new state is generated, avoiding memorizing of duplicates.*

1.4.1 Hash Based Duplicate Detection

To achieve a fast duplicate detection, also facing the rising amount of RAM available in today's systems hashing can be used.

Definition 18 (Hash Function) *A* hash function h *is a mapping of some universe* U *to an index set* $[0, \ldots, m-1]$.

The set of reachable states S is a subset of U, i. e., $S \subseteq U$. Since S is usually not known prior to the search, the hash function h is defined over all elements in the universe. The upper bound for $m - 1$ is the number representation in the system being 2^b where b is the number of bits used to store the value. A generated state s or its representation is stored at a specific position in a table. Usually the predefined position for s is $h(s) \mod$ *tablesize* where *tablesize* is the maximal number of elements the table can host.

Collision affected hashing

Since not only the hash value is limited but also the number of entries in the table *collisions* will appear. A collision appears when two different states are assigned to the same entry in the table.

$$s_1 \neq s_2 \text{ and } h(s_1) \bmod ts = h(s_2) \bmod ts$$

The appearance of collisions can not be avoided, unless all elements which will be hashed are known prior to the hashing, collision resolving strategies were invented. While this work will present the most common in the following, the interested reader is directed to Knuth (1973).

Chaining A basic strategy to resolve collisions is to store more then one element in one table entry by increasing the size of each table entry and decreasing ts. Since increasing the size of an entry increases the number of collisions by decreasing the ts an alternative is to store a list of elements at each entry and record the number of inserted elements.

Open addressing In this strategy the element is stored at an alternative position if its preferred position is occupied. Storing the element simply at the next free position, denoted as *linear probing* requires a scan up to the next free position for each lookup and increases the lookup time so alternatives like *quadratic probing* exist to decrease this drawback. Here the element is stored on an alternative position computed from the hash value $h(s)$ e. g., $h(s)^2$ in *quadratic probing*.

Cuckoo hashing A strategy with a guaranteed constant lookup time is *cuckoo hashing* (Pagh and Rodler, 2001) where two tables t_1 and t_2 and two hash functions h_1 and h_2 are utilized. The element is inserted either in t_1 using h_1 or in t_2 using h_2. When both entries are occupied an element is removed in one table and the new one inserted. The inserting process restarts with the removed element. The drawback is the possibility to meet circles which can be destroyed by changing the hash functions and rehashing all elements.

Bloom Filer In the *bloom filter* or *Bitstate hashing* (Bloom, 1970) the entry in the table is solely one bit. Checking if state s was already expanded boils down to look up the bit at index $h(s)$ and discard s if the bit is enabled. Otherwise the bit is changed to be enabled. Colliding states are discarded. This strategy provides a maximal compression per state but is not complete due to omitting unexpanded states.

Perfect, non collision affected hashing

Perfect hashing (Botelho *et al.*, 2007) is a space-efficient way of associating unique identifiers to states. It yields constant random access time in the worst-case. Perfect hash functions are bijective. In certain search environments perfect hash functions can be used with a bit vector to compress each state to only one bit in $Closed$. Such an environment can be either a search where all reachable states are known before the search, here a collision free hashing function can be constructed or where all reachable states are determined in an additional step performed before the actual search starts. To construct a perfect hash function according to Botelho and Ziviani (2007), it is necessary to generate the entire state space graph. For the search using a bit vector $Closed$ list certain characteristics of hash functions have to be defined.

Definition 19 (Perfect Hash Function) *A hash function* $h : U \to [0, \ldots, m-1]$ *is* perfect*, if for all* $s \in S$ *with* $h(s) = h(s')$ *we have* $s = s'$*.*

Given that every state can be viewed as a bit vector, and, in turn, be interpreted as a number in binary, a simple but space-inefficient design for a perfect hash function would be to use this number as a hash value.

Definition 20 (Space Efficiency) *The space efficiency of* h *is the proportion* $\lceil m/|S| \rceil$ *of available hash values to states.*

Definition 21 (Minimal Perfect Hash Function) *A perfect hash function* h *is* minimal *if its space efficiency is* 1.

A minimal perfect hash function is an one-to-one mapping from the state space S to the set of indexes $\{0, \ldots, |S| - 1\}$, i. e., $m = |S|$. In contrast to open-addressed or chained hash tables, perfect hash functions allow direct-addressing of Bitstate hash tables. This allows compressing the set of visited states without loosing completeness. The other important property is given if the state vector can be reconstructed given the hash value, which allows to also compress the list of frontier states $Open$.

Definition 22 (Reversible Hash Function) *A perfect hash function* h *is* reversible *or* invertible*, if given* $h(s)$*, the state* $s \in \mathcal{S}$ *can be reconstructed. A reversible minimum perfect hash function is called* rank*, while the inverse is called* unrank.

Of course every perfect hash function can be modified to be a reversible hash function by storing the states in the order given by the hash function in a table. Now, when $h(s)$ is computed a lookup suffices to find the reconstructed state but this approach is not efficient in terms of memory usage.

While the generation of a minimal perfect hash function from a set of elements is described in detail by Belazzougui *et al.* (2009) it can be also constructed combining not perfect but *orthogonal* hash functions.

Definition 23 (Orthogonal Hash Functions) *Two hash functions* h_1 *and* h_2 *are* orthogonal*, if for all states* s*, and* s' *with* $h_1(s) = h_1(s')$ *and* $h_2(s) = h_2(s')$ *we have* $s = s'$*.*

Theorem 1 (Orthogonal Hashing implies Perfect Hashing) *If two hash functions* $h_1 : U \to [0, \ldots, m_1 - 1]$ *and* $h_2 : U \to [0, \ldots, m_2 - 1]$ *are orthogonal, their concatenation* (h_1, h_2) *is perfect.*

Proof. Starting with hash functions h_1 and h_2, let s be any state in U. $(h_1(s), h_2(s)) = (h'_1(s), h'_2(s))$ implies $h_1(s) = h_1(s')$ and $h_2(s) = h_2(s')$. Since h_1 and h_2 are orthogonal, this implies $s_1 = s_2$. □

In case of orthogonal hash functions, with small m_1 the value of h_1 can be encoded in the file name (Korf, 2003a), leading to a partitioned layout of the search frontier, and a smaller hash value h_2 to be stored explicitly. Orthogonality can cooperate with bit vector representation of the search space, as function h_2 can be used as an index. For frontier search, the space-efficiency is smaller than with full state space memorization.

Considering its strong set of assumptions of orthogonal and reversible hash functions, hash-based delayed duplicate detection based on perfect hashing as proposed by Korf and Schultze (2005) is not available for general state space search.

1.4.2 Sorting Based Duplicate Detection

In searches where the internal memory does not suffice to expand the whole graph external memory can be used. In this strategy sorting reduces the number of read accesses to the external media since two sorted files can be compared by one parallel scan through them. Figure 1.7 visualizes this process. One file is stored in the buffer while the other is traversed sequentially comparing each state in memory s_m to one external state s_e. The decision process is defined as follows:

- $s_m = s_e$ remove s_m from memory
- $s_m < s_e$ move the memory read pointer
- $s_m > s_e$ move the external read pointer
- $s_e <$ (*following adjacent* s_e) move the memory read pointer to the beginning

Where the comparison is defined by the sorting function.

Since checking for the existence of a duplicate immediately would require a block access to the disk, this is postponed and all elements in one block are checked at once in external searching.

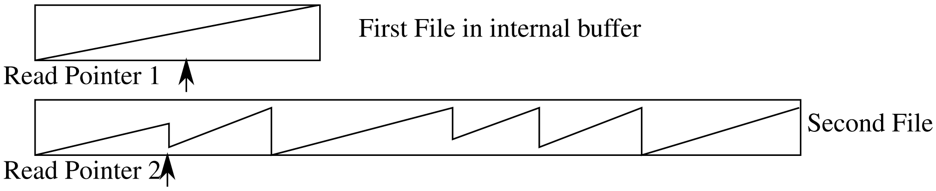


Figure 1.7: Removing duplicates in two sorted files by scanning. Read Pointer 1 is moved to the right and reset to the front when Read Pointer 2 detects a decrease in the increasing sorting order.

1.5 Main Contributions

The underlying publications of this work are the results of intensive discussions not only with my supervisor Stefan Edelkamp but also with a number of scientists in- and outside the office. Since a diary, with detailed results of discussions or even paper dependent contributions does not exist, I can map my contributions to specific chapters and sketch my impact on the results, only. One thing to mention is that the order of author names on the publications is strictly alphabetical. An exception is (Sulewski *et al.*, 2011) here the names are ordered by the amount of contribution.

Although this work relays on a number of publications it extracts common topics from different disciplines and presents the results in a novel, easy to follow way. The author contributes with this thesis to four scientific disciplines by exploiting aspects found in all of them and by proposing a way of using novel hardware efficiently.

Explicit State Model Checking The starting point of utilizing solid state disks in the field of search was the algorithm from Gastin and Moro (2007). I analyzed the *External Perfect Hashing* (Botelho and Ziviani, 2007) which generates the Perfect Hash Function from external media but stores it internally, and developed an efficient strategy to outsource the hash function to SSDs. This externalization enabled the semi-external algorithm presented in Chapter 6 which stores solely one-bit per state in the $Closed$ list in internal memory.

The demand to generate the entire state space efficiently for the previous algorithm motivated us to search for hardware to accelerate the generation. This search led to graphics cards due to the enormous computation power. I analyzed the Model Checking process for entry points to use the graphics unit and extended the existing model checker DIVINE to generate the state space utilizing the GPU. For this the model is converted into the Reverse Polish Notation (RPN). A representation revealed by me to be efficient due to the flat representation and sequential evaluation. To accelerate the generation I evaluated the sorting strategies mentioned in Part I and developed the approach presented in Subsection 4.3.1 as a combination of *MP5* and hash based bucket sorting, efficient for sorting large elements on the GPU.

During the work on sorting, the model checker DIVINE has been reimplemented and ported to utilize 64-bit architecture. An integration of the GPU would have required serious analyzing efforts in DIVINE without the guarantee that further major changes in the code would not make the efforts useless. So the decision was to to develop a CUDA *Driven Model Checker* (CUDMOC) as an evaluation platform for GPU Model Checking algorithms. It has been entirely developed and implemented by me.

Action Planning Inspired by the results in Model Checking and due to the similarity of searching a state space I implemented a parser for the Planning Domain Description Language (PDDL) and extended CUDMOC to the CUDA *Driven Planner* (CUDPLAN). Here the main task was to translate the input into the RPN, enabling the planner to handle action costs by performing a Dijkstra search. The resulting algorithm developed and implemented by me is presented in Part III.

Game Solving In the work on Model Checking and Action Planning a byproduct was the *successor counting* strategy proposed in Subsection 4.2.1. It revealed to be inefficient in both disciplines. This strategy was evaluated by me on single player permutation games and revealed to be efficient in Game Solving. During the evaluation I reduced the space consumption of the two-bit BFS approach from Cooperman and Finkelstein (1992) to use only one-bit for reachability or even for BFS in special cases, presented in Part IV. In this context Perfect Hashing for permutations, called *ranking* was utilized to compress the states. I revealed the lexicographical approach is inefficient on the GPU and it is a better solution to use the functions proposed by (Myrvold and Ruskey, 2001) adopted to be efficient on the graphics card. The approach was extended to single and multi-player games by using binomial and multinomial hashing.

Discussions with other scientists revealed the field of Probabilistic Model Checking to be a promising target for a GPU enhancement. It turned out that the Jacobi iterations used to solve the probabilistic problems are efficiently parallelizable. So I analyzed the implementation of the PRISM Model Checking tool and extended it to use the GPU and later on even multiple GPUs as presented in Part V.

Although most of the work done by me seems to be implementation I also participated actively on writing the papers.

1.6 Organization of the Thesis

This thesis presents results achieved by utilizing the recently introduced achievements in information storage and parallel processing on four scientific areas from Software Verification and Artificial Intelligence domains, namely *Explicit State Model Checking*, *Action Planning*, *Game Solving* and *Probabilistic Model Checking*. All relies strongly on traversing a state space given by an initial state and an transition relation.

Having introduced the basic formalism of state space searching and basic algorithms in the first chapter it continues with a brief overview of the hardware evaluation in the last decade. Since knowledge of the specifics is crucial to understand the decisions undertaken in the development of the algorithms Chapter 2 introduces the main aspects of *Solid State Disks* (SSDs), followed by the architecture and the programming model of *General Purpose Graphics Processing Units* (GPGPUs). The chapter closes with discussing the limitations in GPU programming.

After presenting the fundamental knowledge on the hardware an algorithm is proposed to utilize both devices efficiently to extend the number of solvable problems in the search domains. The presented algorithm is build up of modules which are used depending on the problem. The searching process is split up in three stages, namely *evaluate successors*, *generate successors* and *remove already existing successors*. In each stage strategies are presented for an efficient utilization of the parallel processors of the graphics card or the random access speed of SSDs. To avoid revisiting of states two strategies are explored, on the one hand an external sorting approach which utilizes the GPU to speed up sorting, on the other hand CPU based hashing. Both strategies are extended to increase the available space per element by compressing and adding the probability of removing unexpanded states.

Having introduced Explicit State Model Checking the proposed strategies are applied and evaluated. To extend the available memory for the checking process techniques to use Solid State Disks as a storage medium are studied in Chapter 6. Here, a semi-external technique to use perfect hash functions stored externally is proposed to find minimal counter examples in LTL Model Checking and concentrates on the results from the following journal publication,

- Stefan Edelkamp, Damian Sulewski, Jiri Barnat, Lubos Brim, Pavel Šimeček, *Flash memory efficient LTL Model Checking*. In Science of Computer Programming, volume 76, number 2, pages 136-157, Elsevier, 2011

Having extended the available memory by using external media we increase the speed of checking by enhancing the process with a massively parallel graphics processor in Chapter 7. Algorithms to speed up breath first searching by using the computation power of a GPU are presented. Detailed information is given on translating the given problem into a description suitable for the graphics card, and the implicit traversal of the graph. This chapter merges the results from two publications, namely

- Stefan Edelkamp and Damian Sulewski *Efficient Explicit-State Model Checking on General Purpose Graphics Processors*. In 17th International SPIN Workshop on Model Checking of Software (Spin'10) by van de Pol and Weber (Eds.). Lecture Notes in Computer Science (LNCS), vol. 6349, pages 106-123, Springer, Berlin, Heidelberg, 2010
- Stefan Edelkamp and Damian Sulewski *External Memory Breath-First Search with Delayed Duplicate Detection on the GPU*. In Sixth Workshop on Model Checking and Artificial Intelligence (MoChArt'10), Atlanta, Georgia, USA, July 11, 2010

Due to the similarities in Model Checking and Planning, the approach presented in the previous chapter is adopted in Action Planning by an implementation of a parser and the addition of a cost-optimal search technique. Chapter 9 introduces planning and points out the similarities to Model Checking. In the following the conversion of a planning problem to a GPU suitable description is presented. The approach was extended to Dijkstra's Algorithm, where the costs of a node are computed on the GPU, and presented in

- Stefan Edelkamp, Damian Sulewski and Peter Kissmann *Exploiting the Computational Power of the Graphics Card: Optimal State Space Planning on the GPU*. In International Conference on Automated Planning and Scheduling (ICAPS'11), pages 242-249, AAAI Press., 2011

During the evaluation of techniques for Model Checking and Action Planning a different approach for generating successors was found, not suitable for the analyzed domains. This technique showed promising results in areas where the generation of successors is an easy to perform task and even perfect hash functions can be used. Part IV describes the utilization of the GPU for solving games. After an introduction is given, the analyzed games are introduced. A well defined state description, e. g., given by a game board and the number and position of pins of each player, allows us to construct perfect hash functions, which are collision free, and efficiently computable on the

GPU. Chapter 13 will propose such hash functions for the analyzed games which will be used to construct algorithms for traversing the state space. The Chapter is followed by a detailed instruction on porting the algorithms on the GPU and an experimental evaluation presenting the results. It merges the results from the following publications:

- Stefan Edelkamp and Damian Sulewski *Parallel State Space Search*. In The 2009 International Symposium on Combinatorial Search (SoCS'09), Los Angeles, USA, July 2009
- Stefan Edelkamp, Damian Sulewski and Cengizhan Yücel, *Perfect Hashing for State Space Exploration on the GPU*. In International Conference on Automated Planning and Scheduling (ICAPS'10), pages 57-64, AAAI Press., 2010
- Stefan Edelkamp, Damian Sulewski, Cengizhan Yücel, *GPU Exploration of Two-Player Games with Perfect Hash Functions*, In The ICAPS 2010 (International Conference on Automated Planning and Scheduling) Workshop on Planning in Games, Toronto, Canada, May, 2010

Although the proposed framework could be also applied to speed up the state space generation in Probabilistic Model Checking it turns out to be ineffective in this discipline to generate the state space. Here the checking process is performed by solving linear equations so a different approach had to be invented to profit from the GPU. Part V, exemplifies the usage of the GPU to speed up the Jacobi iterations used in Probabilistic Model Checking for solving linear equations, the most time consuming part of the process. It is based on the journal publication

- Dragan Bosnacki, Stefan Edelkamp, Damian Sulewski and Anton Wijs, *Parallel Probabilistic Model Checking on general purpose graphics processors*, In International Journal on Software Tools for Technology Transfer (STTT), volume 13, number 1, pages 21-35, Springer, Berlin, Heidelberg, 2011

Finally, conclusions are drawn and future extensions discussed.

Chapter 2

Hardware and Programming Models

This chapter motivates the necessity for the algorithm designers to follow the hardware development and adopt existing, or even invent new algorithms to utilize the aspects of novel hardware. In graph searching, like in most computationally challenging tasks, not only the computing speed has to be optimized but also the storage capacity. With an increasing computation speed the amount of generated information in given time is increasing, resulting in a larger space demand or better compression techniques. On the other hand the increasing amount of storage capacity enables a longer computation, accelerated by faster hardware or by better algorithms.

In the following novel hardware solutions to both problems will be presented. On the one hand the SSDs are in the process of replacing the HDDs in high performance computing. On the other hand the utilization of massively parallel systems to support the multi-core CPUs. After sketching the evolution of the hardware up to the current point, the chapter closes with a presentation and evaluation of the used devices.

Structure of the chapter: This chapter will sketch the hardware developments of storage and display devices in recent years. Detailed information is given on the differences between magnetic and solid state media and on the interns of graphics processing units. The chapter closes with an evaluation of the hardware used in this thesis.

2.1 Information Storage

The demand of storage capacities rises linearly with the number of states to visit, not only a visited list has to be maintained, but also the states being evaluated in the future have to be stored. This requests a huge amount of storage space necessary to find a goal, or to prove its absence. Traditionally two main levels of storage are distinguished. Internal storage, accessible very fast, even when accessed randomly, and external storage preferring sequential access to deliver data at a decent speed. In the recent years a new type of external storage was presented, the so called *Solid State Disk* (SSD). This drive lacks any mechanical parts and stores its information like the internal storage on memory cells in the chip. Such a type of memory is faster, compared to *hard disk drives* (HDDs) in random reading time but the disks analyzed in this work gained no speed advantage while writing.

2.1.1 Random Access and Insufficient Space in Internal Memory

Even with 64-bit technology the amount of internal storage is limited due to the capabilities of the hardware. Utilizing more then 32 gigabyte of internal random access memory (RAM) assumes special hardware which is very expensive. Even in hardware developed for server usage 256 GB RAM per computing node seams to be the limit nowadays. While connecting a huge number of computers with a fast network is a solution, using them implies different algorithms switching the topic to parallel computing.

2.1.2 Pushing Space Constrains by Going External

While writing this thesis the price for 1 GB external storage on hard disk drives is 0.03 €, making it very cheap to construct cluster computers with nearly unlimited space, or adding storage capacities to existing computers. Comparing the speed of data access on external and internal memory bares the main drawback of HDDs. Even with a sequential data transfer rate of 200 MB/second the disk is slow compared to 17, 000 MB/second achievable with recent RAM modules. On the algorithmic side new traversal techniques developed to access data in large blocks utilizing the sequential access speed of this media can be used. On the hardware side it is possible to connect more than one media to a *Redundant Array of Independent Disks* (RAID). While such an array theoretically increases the transfer speed up to a sum of all used drives it does not change the IO latency when accessing the media randomly.

2.1.3 Solid State Disks

SSDs developed recently benefit from the fact, that accessing data stored in *solid-state memory* of a chip is faster then accessing it on a magnetic disk. They are not only faster while sequential reading but also while random reading. In recent years a trend arises to replace HDDs in notebook computers by SSDs and they are continually entering the market for desktop computers particularly when a high demand for fast access exists. While an SSD is preferred in portable computing due to their resistance to physical shock and the low power consumption, desktop users profit mostly from the speed and the absence of sound. However, the price of 2 € for 1 GB nowadays increases not only the price of portable computing but also decreases the storage capabilities compared to HDDs. Taking into account the already long evolution of magnetic media, and the fact that the production of memory chips is developing very fast the price is expected to decrease in near future[1].

[1] When my research begun two years ago the price was nearly 20 € for 1 GB.

Storage Medium

The drawback of magnetic and optic devices is the time needed to move the read/write head to the data. To eliminate this drawback early generation solid state devices stored the data in electrically erasable DRAM chips which were powered by additional batteries if necessary. Of course the data is lost once the battery drains out. In 1994 the first flash-based drive was introduced which did not require an additional power supply. The military begun to use such devices due to their physical shock resistance. Recently the manufacturers switched from using DRAM volatile memory to NAND flash due to its higher bit density (Kim *et al.*, 2005) and lower production cost reaching up to 1 TB capacities with a throughput of 768 MB/s. Table 2.1 shows a qualitative picture for the devices tested by (Ajwani *et al.*, 2008).

Table 2.1: Rough classification of flash with respect to RAM and hard disk.

Characteristic	RAM	Flash	Disk
Volatile	Yes	No	No
Shock Resistant	Yes	Yes	No
Physical Size	Small	Small	Large
Storage Capacity	Small	Large	Largest
Energy Consumption	High	Low	Medium
Price	Very High	Medium	Very Cheap
Random Reads	Very Fast	Fast	Slow
Random Writes	Very Fast	Fast	Slow
Sequential Reads	Very Fast	Fast	Fast
Sequential Writes	Very Fast	Fast	Fast
Throughput	Big	Smallest	Small

Comparison to Magnetic Media

While a magnetic drive is mostly a mechanical device, where one or more rotating magnetic disks are accessed by a read/write head which is moved to the according position, an SSD is an electronic device where no moving parts are present. Figure 2.1 points out the aspects of both drives in more detail. The controller's task in the hard disk is to activate the motor spinning the disk and to move the head to the appropriate location when accessing data. The data is stored in continuous blocks if possible. Although NAND flash memory is slower it has outpaced DRAM memory due to the highly parallel structure of the SSD and the evolution of on board controllers built into the devices. The controller in an SSD is responsible for accessing the appropriate memory cell and to distribute written data over the available blocks. While a block on the magnetic disk can directly be overwritten this is not the case in NAND memory. Here, the controller has to erase the block prior to writing data to it, a task which explains the discrepancy between reading and writing times when accessing solid state drives.

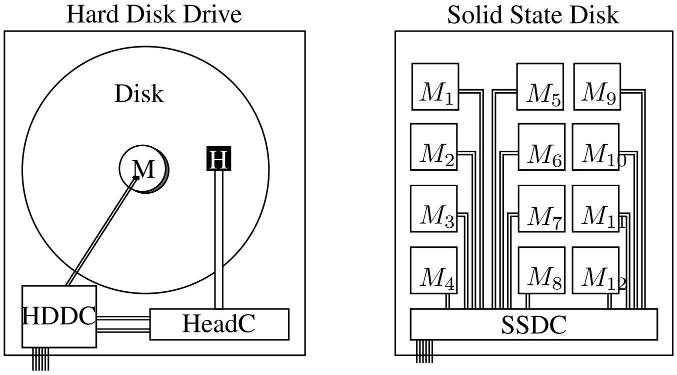


Figure 2.1: Comparison of a HDD and an SSD on the hardware level. In a HDD the controller (HDDC) controls the motor (M) to spin the disk (Disk) and the head controller (HeadC) to move the head (H). Data is read and written sequentially using the head to the disk. In an SSD the controller (SSDC) accesses the memory chips ($M_{1...12}$) directly receiving and sending data in parallel from them and to them.

Memory Models

In this section an extension to the model for hard disk I/O operations presented by Aggarwal and Vitter (1988) is sketched (given in detail by Edelkamp and Sulewski (2008a)), and extended to represent also the characteristics of solid state drives. It also distinguishes between the scanning complexity $\mathit{scan}(n) = \lceil n/B \rceil$ and sorting complexity $\mathit{sort}(n) = \lceil n/B \rceil \log_{\lfloor M/B \rfloor} \lceil n/B \rceil$, where n is the number of input items that have to be scanned and sorted, B is the number of elements in a block that are accessed in one I/O operation, and M is main memory capacity (in items).

Driven by own evaluation and the observations of Ajwani *et al.* (2008), this model distinguishes between writing of n items, denoted as $\mathit{write}(n)$, and reading of n elements, denoted as $\mathit{read}(n)$, mainly because reads are faster than writes. As, according to Ajwani *et al.* (2008), standard external sorting does not differ much on both media introducing a different term for flash memory is not necessary. It has to be accepted that the derived complexity model is not an exact match. For example there are discontinuities on flash media, e. g., restructuring the access tables requires longer idle times.

Solid state disk reads also operate block-wise, as reads and writes on hard disk do, so reading small amounts of data from distant random positions takes considerably more time than reading the same amount of data stored linearly. This does match the design of flash media. The difference in read and write on flash devices have to reflect the fact that, since NAND technology is not able to write a single random bit, writing uses block copying and removal, before a small amount of data is written[2]. This explains why random writing is slower then random reading.

As observed by Ajwani *et al.* (2008), a simple penalty factor is likely to be too

[2] The gap between read and write also relates to how the flash memory is organized. Even for NAND devices different trade-offs can be obtained in different designs.

pessimistic for random write, as it becomes faster if the data set gets larger. The theoretical model suggested here is based on linear functions that include both the offsets to prepare the access to the data, and the times for reading it. As hard disk I/Os as well as read and write I/O operations differ, it suggests not to count block accesses, but take some abstract notion of time. The proposed model distinguishes between the offset (including the seek time and other *delays* prior to the sequential access), and a linear term for reading the elements.

One may either introduce individual block sizes for reading and writing, or devise suitable factors for read and write access. The second alternative is preferred here and suggests the following primitives for analyzing algorithms on flash memory:

$$\begin{aligned} \mathit{scan}(n) &= t_A + t'_A \cdot \lceil n/B \rceil, \\ \mathit{read}(n) &= t_R + t'_R \cdot \lceil n/B \rceil, \text{ and} \\ \mathit{write}(n) &= t_W + t'_W \cdot \lceil n/B \rceil, \end{aligned}$$

where the value

$\boldsymbol{t_A}$ denotes the time offset for hard disk access (either for reading or for writing). It reflects the seek time for moving the head to the location of the first data item that is executed only once in a sequential read operation.

$\boldsymbol{t'_A}$ is the time for reaching the next block in a linear scan.

$\boldsymbol{t_R}$ **and** $\boldsymbol{t_W}$ are the time offsets for reading and writing data on the flash. With t_W being considerably larger than t_R this explains the discrepancy between the two operations for random access.

$\boldsymbol{t'_R}$ **and** $\boldsymbol{t'_W}$ are the offsets for flash media access per block. Here t'_W is only moderately larger than t'_R, explaining that the burst rates do not differ that much.

As t_A is much larger than t_R and t_W (while t'_A is about as large as t'_R and t'_W) these terms agree with the observation that for flash media it is less important, where the block is located, while disks clearly prefer blocks that are adjacent.

Ajwani *et al.* (2009) propose two new computation models, the general flash model and the unit-cost model being enough for meaningful algorithms design and analysis picking up the idea presented here and simplifying it for easier usage.

This section presented the organization of semiconductor storage devices and compared their function to magnetic media devices. The most limiting factor of SSD usage is the limited space. While hard disk drives can store several terabytes today, available SSDs are limited to several hundred gigabytes. A second negative aspect is the immense influence of the controller. Different controller technologies with varying access speeds result in different running times of algorithms. Additionally, when analyzing an algorithm the developer has to distinguish between read and write accesses, compared to just counting the IO operations in the Vitter/Shriver model.

2.2 Faster Computation Using Parallel Hardware

In his thesis Jabbar (2008) proposed algorithms to utilize external hardware for state space exploration in a way which hides the latency of the external media, relying on more computational power for a faster search. The experimental evaluation was performed on a two disk RAID array with nearly 120 megabyte per second transfer rate. Combining four recent SSDs in such an array enables nearly 1,000 megabyte per second transfer rate for external media, pushing the bottleneck even further on the central processing unit.

The increasing speed of external memory and also the perspective to utilize more internal memory motivated researchers to develop parallel algorithm to solve harder problems.

2.2.1 Parallel Computing

In the 90ties Gasser (1996) used a cluster of connected computers to solve the Game Nine-Men-Morris. Utilizing connected machines enabled the researchers to tackle complex problems and get deeper insight into solutions. By this time two types of parallel computers existed, a distributed system of self-sufficient machines connected with a network and systems consisting of several unique CPUs with access to shared memory. *Cluster* computers combine both architectures to achieve more computation power.

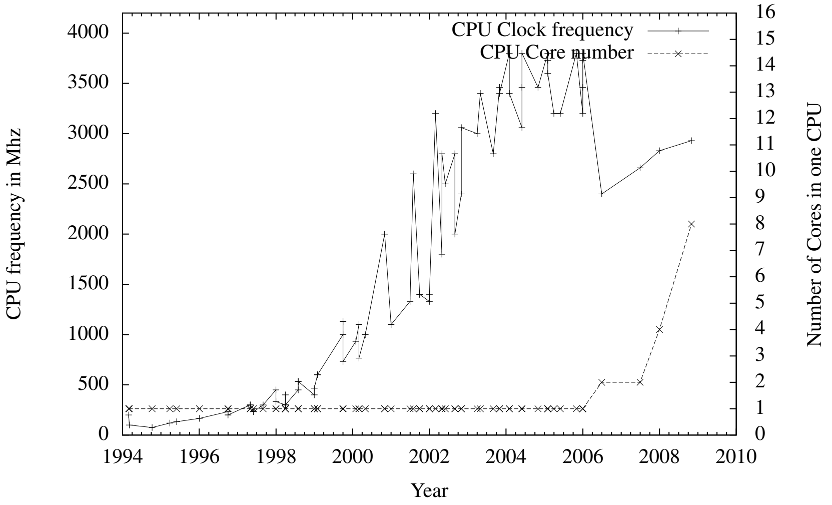


Figure 2.2: Intel CPU frequency over the years.

In 1965 Moore predicted the number of transistors placed inexpensively on a chip to double every two years. Until nearly 2005 not only the number of transitions on a chip nearly doubled every two years, following "Moore's Law" but also the frequency of the chip doubled. According to Figure 2.2 this trend stagnated in 2005 where the developers started to double the number of cores every 2 years. The high frequency scaling, attended by a rapid development in network technology, took the focus from machines with more then one CPUs on one main board and pushed it to clusters of single core machines connected in a high speed network. This development achieved a peak in the year 2000 with the fastest computer in Europe consisting of 528 single standard PCs, each with just one Intel Pentium III CPU.

Since running the processors at a very high frequency causes the CPU to consume much power and cooling becomes problematic, a way to increase computational power without increasing the clock speed had to be found, and the solution led to multi-core CPUs. Here several computing cores are combined in one chip and each single core runs at a relatively slow clock speed, but the computational power can be cumulated. Today multi-core CPUs consists of up to 8 real cores simulating 16 cores by *Chip-Level-Multithreading* (CMT).

The development in architecture shifted the focus in algorithm engineering again from distributed algorithms communicating over a network to multi-core algorithms communicating using shared memory.

2.2.2 General Purpose Graphics Processors

While the cores of a CPU are self-sustaining cores being able to accomplish computation tasks independently the developers of Graphics Processing Units (GPUs) being confronted with image processing choose a different architecture for their many-core chips. In image processing where the same function has to be computed for a huge number of aspects in the image, an *Single Instruction Multiple Data* architecture is preferred. Here a huge number of processors executes the same instruction on different data. Usually no synchronization is necessary and the processors identify their chunk of data using an unique id.

Lets take a look into the past to motivate the development of the graphics hardware. In the early days of computing the only purpose of displays was to visualize text. Some rather simple components, located directly on the Motherboard were enough to accomplish this task. The *graphics card* was born in the early 1980s, when IBM moved the hardware responsible for data visualization to a separate card, plugged into a bus and equipped with own memory. The evaluation of computing power demanded for higher standards in data visualization. Table 2.2 enumerates the standards sorted by time of definition. The resolution of the MGA standard is defined in graphics blocks which can display a character each, the remaining resolutions describe the amount of pixels per row and column.

Starting in the 1990s the manufacturers constantly added more memory onto the graphics card allowing to increase the resolution and the amount of colors per pixel.

The CPU computed the visualization and the graphics card displayed it until a *graphics accelerator chip* was added onto graphics cards in 1994. From now on an increasing number of tasks necessary to display an image on the screen were performed by the accelerator and its complexity and computational power rapidly increased with the demands of 3 dimensional (3D) data visualization. The *Graphics Processing Unit* (GPU) was born. Eickmann (2004) presents a detailed history of graphics cards.

Table 2.2: Graphics standards over the time.

Year	Name	Memory	Resolution	Colors
1981	MGA	4 KB	80x25	Monochrome
1982	HGC	16 KB	720x348	3 Colors
1982	CGA	16 KB	320x200	4 Colors
			640x350	2 Colors
1985	EGA	128/256 KB	320x200	16 Colors
			640x350	16 Colors
1986	VGA	512 KB	320x200	256 Colors
			640x350	16 Colors
...				
2011	UHXGA	4 GB	7680x4800	2^{48} Colors

In 1999 NVIDIA presented a GPU being able to visualize a 2 dimensional (2D) projection on the screen given a model of a 3D world. It was capable to display changing viewer positions and different light modes denoted as *Transformation and Lightning* (T&L). From then on, the GPU becomes faster and supports more and more functions and transformations for image processing and formatting. Since image processing is beyond the scope of this work only the existence of APIs like OpenGL[3] which enable the programmer to use instructions supported by the given GPU is mentioned here.

The task of projecting a 3D world on a 2D screen consists of a huge number of independent computations, e. g., computing the color for each polygon, and the GPU evolved to a highly parallel processor being even superior to the CPU in certain tasks. Therefore it was a natural process when in 2003 the first thoughts came up to utilize the GPU not only for data visualization, but also for data processing.

Summarizing can be said that today's graphics cards contain a graphics processing unit and memory being used exclusively by the GPU. A special case are graphics cards which are located on the Motherboard and share RAM with the CPU.

For understanding the decisions which lead to the proposed algorithms, knowing the utilized hardware is essential. Thus, this Chapter sketched the evolution from character displays used in the early 1970s to visualize text, to nowadays graphics cards being not only capable to visualize data but also to perform highly parallel computations. A detailed insight into current hardware is given, followed by an introduction into the different programming interfaces with the main attention given to CUDA, the interface from NVIDIA.

2.2.3 GPGPU Programming Interfaces

Programming languages do not support additional hardware like GPUs natively, they have to be extended by additional instructions provided by a programming interface. For general-purpose computing on graphics processing units (GPGPU or GP^{2}U) a general purpose API is not necessarily needed, in the beginning the data was interpreted

[3] see: `http://www.opengl.org/`

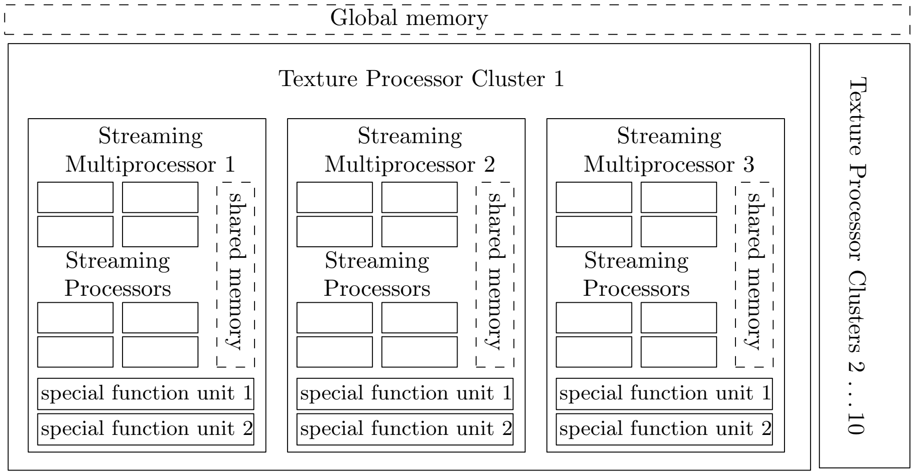

Global memory
Texture Processor Cluster 1
Streaming Multiprocessor 1
Streaming Multiprocessor 2
Streaming Multiprocessor 3
shared memory
Streaming Processors
special function unit 1
special function unit 2
Texture Processor Clusters 2 . . . 10

Figure 2.3: Sample Architecture of the NVIDIA GTX 280 chipset. The GPU consists of a number of Texture Processing Clusters (TPC) connected to the global memory. Each TPC includes several Streaming Multiprocessors (SM) which are composed of Special Function Units, responsible for complex computation tasks and work distribution, shared memory and a number of Streaming Processors (SP). SP are simple computing devices being used by the SM as SIMD processors evaluating the equal instructions in parallel.

by the GPU as image data and operations on it had to be formed using the graphical instructions supported by OpenGL or DirectX. As this is inconvenient for the programmers, general-purpose APIs had to be invented.

Architecture

When this work began CUDA was the only manufacturer who delivered a programming interface suitable for scientific research. This work is therefore based on the CUDA architecture from NVIDIA utilizing the CUDA SDK. Furthermore only one GPU could be afforded for experimental evaluation and the NVIDIA GPUs supplied more computation power and on-board memory. Therefore the next sections describe CUDA specifications but also point out the similarities and differences to the Stream SDK from ATI.

Instruction evaluation In image processing the same function is executed on many data entries in parallel. The given world is decomposed in triangles, the more complex an object is, the more triangles are needed to display it accurately on the screen. Hence, when the image is computed, the GPU has to perform the same computation, e. g., display light conditions, visibility or surface visualization for each triangle, independently. Such a computation is fairly easy to parallelize by using several computation nodes and allowing them to access the data at fast speed. Since the tasks are independent and homogeneous, the computation *speed-up*[4] is linear to the number of cores used. To meet the demand of faster image processing and visualization of complex objects build up of millions of triangles the manufacturers had to increase the number of processing units in the GPU. Other aspects of general computing architecture, e. g., fast randomized access to memory or even autonomous computation of several cores, are not necessary on a graphics card.

This development currently peaks in an architecture like the one in Figure 2.3, being representative for GPUs nowadays. A NVIDIA GPU consists of 10 *texture processor clusters* TPCs being responsible for access to the data and the instructions from the host and for distributing the instructions to its *streaming multiprocessors*(SM). Each of the SMs in a TPC can be seen as an autonomous core with an internal shared memory and various computation nodes. Two *special function units* SFUs perform double precision arithmetics and higher mathematical functions like sine and cosine. Instruction evaluation is done by the 8 *streaming processors* (SPs) included in each of the SMs. The SPs are *single instruction multiple data* SIMD cores executing exactly the same instructions in parallel.

The Architecture of ATI is similar to the one of NVIDIA, here the TPC is called *SIMD Engine*, the SM are *Thread Processors* and the SPs bear the name *Stream Cores*. Since this work is based on the NVIDIA architecture it will use the names of NVIDIA for the components.

Data Management

With a rising number of processors the access speed to the data becomes the bottleneck of the computation. To serve the cores faster with data a graphics card is equipped with dedicated *global memory* (Video RAM or VRAM) which can be accessed from the main system. Data, to be accessible by the cores, has to be copied over the bus from the system memory RAM and back when necessary. The amount of VRAM is currently limited to 2 GB on standard graphics cards and to 4 GB on high end graphics cards.

The hierarchically structured memory on a graphics card is partitioned in three layers, from which only the first one, the VRAM resides on the card while the remaining structures are located in the GPU.

Figure 2.4 visualizes all different memory hierarchies located in a system considered in this work. Since the distribution of information is a crucial part of an algorithm, strategies have to be developed to utilize each layer of memory efficiently.

As described above, the VRAM is located outside the GPU therefore sequential access speed to it is comparable to the sequential access speed to RAM or even faster due to the increased width of the bus. It is optimized for streamed access in blocks, but supports random access also. To maximize the throughput for image processing this memory supports *coalescing* a technique where a number of adjacent memory accesses is combined to one block access by the memory manager. Due to the capability of *broadcasting*, access from multiple SPs to the same memory region is fast, while

[4] defined as one core time divided by many core time.

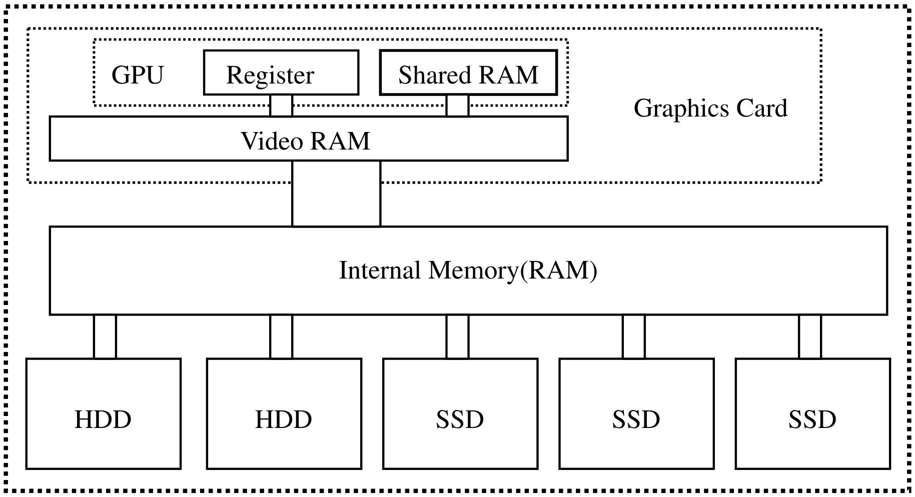


Figure 2.4: Visualization of available memory hierarchies in the system. At the bottom the external devices, being either HDDs or SSDs are displayed and connected directly to the internal memory, the RAM. Access to the graphics cards global memory (VRAM) is established through a BUS on the main board. On each graphics device the GPU accesses the VRAM and manages internal memories like the shared memory and the registers.

randomized access has to be sequentialized.

Shared memory (SRAM[5]) is located in each of the SMs and accessible by all its streaming processors. In current architectures it is limited to 32 KB per SM being accessible at a speed comparable to the cache in a CPU.

Each SP has exclusive access to registers which are accessible at a high speed but very limited in size.

Principally each part of a GPU has its counterpart on a desktop computer, where only the latencies are much higher. The VRAM is fully adequate to a hard disk drive which also prefers sequential access being enhanced by reading blocks of the data. Random access is provided here but punished with high latencies. Each texture processing cluster is mapped to a multi-core CPU on the main board while each multiprocessor is represented by a core of the CPU and, finally a thread, assuming multithreading is supported, describes a streaming processor of a GPU.

While all aspects of a GPU can be mapped to a desktop system the reverse is not the case, a GPU does not have a counter part for the RAM. This aspect points out the difficulty when developing algorithms for such an architecture. Omitting the RAM in a system results in a direct access of (unsynchronized) cores to the hard disk which can end disastrous. The mapping of virtual threads to GPU processors omits the fact that the processors are SIMD which run in parallel while threads are independent from

[5] Not to be confused with the static RAM.

each other and run sequentially.

Even a mapping of a multiple GPU system to a cluster is possible on the memory and also the computation level. Each cluster system has a storage-area-network (SAN) capable of storing large amounts of data. When this SAN, connected over the network, is accessed randomly each access will be paused for a network latency time needed to build up the communication, what is fully adequate to access external media in a desktop system. The VRAM of the GPU is then mapped to the shared memory of each cluster node connected to one or more multi-core CPUs which perform the computation in parallel just like the streaming processors of a GPU.

Having mapped the GPU system to a desktop and even a cluster computing system motivates the development for graphics card supported systems even more due to the reutilization of such algorithms on other computation environments.

Programming Paradigm

The programming model is based on the idea of a kernel driven by threads. The kernel is a function which is executed on up to all SPs located on the GPU. Based on the SIMD idea the kernel is the sequence of *Single Instructions* which is executed on *Multiple Data*.

Algorithm 2.1 shows a sample CPU algorithm to compute the multiplication of a matrix A with the vector V and store the result in the vector R. The CPU implementation would be implemented using two embedded FOR loops traversing each row and column of A, multiplying the given entry with the appropriate entry of V and storing the result in the vector R. This implementation results in a $O(n \times m)$ running time.

It can immediately be seen that the outer loop (increasing the variable i) can be parallelized, since no access to other lines then i is necessary for computing R_i. In GPU programming this algorithm would be divided into two parts, one executed on the host and the other on the GPU. The instructions for the GPU are extracted into the kernel resulting in Algorithm 2.2. Note that an integer i is required in the kernel to specify the line each one has to compute. The corresponding host implementation is similar to Algorithm 2.3. Although a parallel version could also compute the multiplication $A_{ij} * V_j$ in parallel, this would require a kernel invocation for each cell, resulting in an inefficient algorithm due to the small amount of work a kernel has to do in contrast to the overhead when starting it.

Algorithm 2.1: Matrix Vector Multiplication on the CPU

```
Input  : $n \times m$ matrix $A$, vector $V$, integer $n$, integer $m$
Output: vector $R$
for $i = 0$ to $n$ do                                {traverse each line}
    for $j = 0$ to $m$ do                            {traverse each column}
        $R_i = R_i + A_{ij} * V_j$ ;                 {perform computation}
```

Algorithm 2.2: *GPU-Kernel* for the Matrix Vector Multiplication

```
Input  : $n \times m$ matrix $A$, vector $V$, integer $i$, integer $m$
Output: vector $R$
for $j = 0$ to $m$ do                                  {traverse each column}
    $R_i = R_i + A_{ij} * V_j$ ;                       {perform computation}
```

Algorithm 2.3: Host algorithm for the Matrix Vector Multiplication

```
Input  : $n \times m$ matrix $A$, vector $V$, integer $i$, integer $m$
Output: vector $R$
copy data to GPU ;
forall the $i = 0$ to $n$ do in parallel                   {traverse each line}
    $rowVectorKernel(A, V, i, m)$ ;                 {call GPU-Kernel on this line}
copy result to host ;
```

Software Developing Kits

In 2008, by the time this work started, two major graphics card manufacturers ATI and NVIDIA provided a software development kit for the C++ language, the ATI Stream SDK and the NVIDIA CUDA SDK.

ATI Stream™ In November 2006 AMD released a programming interface called *Close To Metal* (CTM) giving the programmers the possibility to access the native instructions of ATI GPUs. Interpreting the name as close to the hardware reveals the high complexity of the interface which made it uninteresting for the majority of general purpose programmers and for scientific evaluation.

Stream SDK[6] displaced CTM in December 2007, replacing the low level instructions by instructions on a higher level. Stream is based on the Brook language developed by the Stanford University (Buck *et al.*, 2004).

In 2010 AMD stops the development of an own SDK and switches over to support the open standard *OpenCL*.

NVIDIA CUDA After AMDs release of CTM, NVIDIA presented an SDK for their *Compute Unified Device Architecture* (CUDA) based graphics cards in February 2007, which supports low level and higher level instructions and has evolved to version 3 today also supporting OpenCL.

Software Hardware Mapping in the CUDA SDK

The GPU has to know the number of kernels to start. This can be determined from the array the kernel is invoked like in the Stream SDK or directly given by the user. In the CUDA SDK the user specifies at the number of kernels, denoted as *threads*

[6] `http://www.amd.com/stream`

when starting the computation by grouping them into *blocks* and arranging them in a 2 dimensional *grid*.

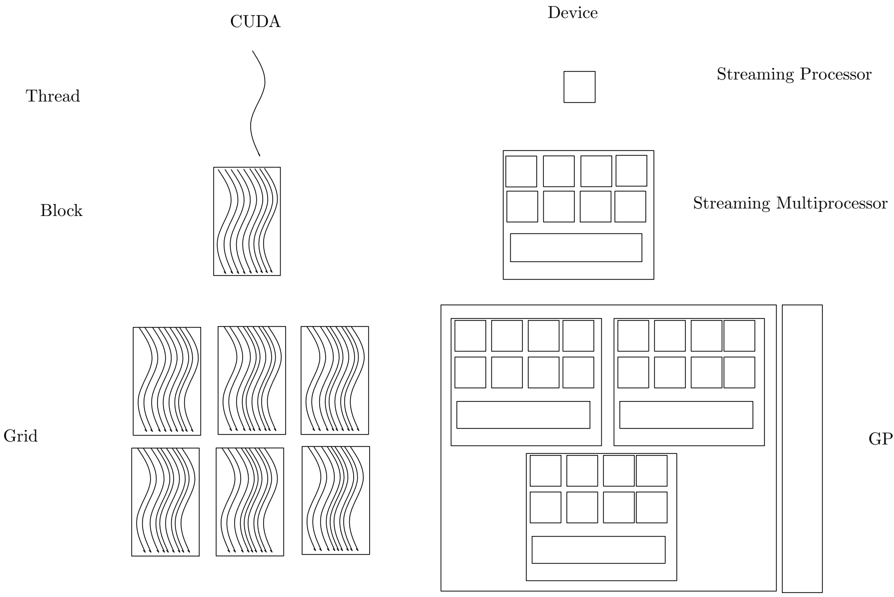


Figure 2.5: Mapping of the software structures to the GPU hardware in CUDA. Each kernel is called a *thread* and up to 512 threads are grouped into one *block*. The number of blocks to execute is given in 2 dimensions and called *grid*. The scheduler decides which block to execute and assigns it to a streaming multiprocessor where each thread is assigned to one streaming processor.

Figure 2.5 sketches the mapping of software structures to the GPU hardware. The TPC schedules the execution of the blocks by assigning them to individual streaming multiprocessors which execute the kernels on their streaming processors. Memory latencies can be hidden by scheduling the kernels which already received the data and pausing the remaining ones.

A CUDA kernel is invoked preceding it with the grid dimensions enclosed by <<< >>>. The grid defines how many threads are grouped together to a block. Each thread corresponds to a kernel invocation. All the threads in a block are scheduled to the same SM allowing them to share the SRAM and being synchronized. Sharing of information between threads in different blocks is only possible using the slow VRAM.

Limitations

A number of limiting factors exists when developing GPGPU algorithms. The key limitation is the slow random access to the VRAM from the processing cores. This prohibits using pointers in data structures. All memory access to the data has to be streamed or at least synchronized for many threads.

Before the GPU code is compiled all functions are concatenated *inline* by the compiler resulting in one sequence of code instructions. While this approach increases the performance it makes using recursion impossible. All recursive functions have to be rewritten to sequential ones before using them on the GPU.

Another limitation is given by the amount of memory incorporated on every streaming multiprocessor. The maximal number of concurrent kernels on one SM is given by its memory divided by the amount of memory the kernel uses. Utilizing to much memory in a kernel will force the SM to execute only a small number of kernels leaving most of the streaming processors idle.

2.3 Used Hardware

For an evaluation of the approaches presented in this work specific hardware settings were used, evaluated in the following sections. After giving an overview on the used solid state disks and comparing them to magnetic drives the specifications of the analyzed graphics cards will be given closing with a detailed evaluation of the different systems.

Two systems were evaluated in the scope of this work starting with a *one SSD, one GPU* 32-*bit* system and continuing with a *two SSDs, two GPUs* 64-*bit* system.

- The 32-bit experiments were executed on (one core of) a personal computer with an AMD Athlon(tm) 64 X2 Dual Core Processor 3800+ running at 2 GHz with 4 GB of RAM. The solid state drive is a HAMA 3,5” SATA device with a storage capability of 32 GB. This system includes an MSI N280GTX T20G graphic card with 1 GB global memory and 240 cores running at 0.6 GHz plugged into a PCI Express slot.

- The 64-bit experiments were executed on (one, or more cores of) a personal computer with Intel Core i7 CPU 920 running at 2.67 GHz providing up to 4 CPU cores. Two SSDs were used here to test grouped configurations on them, one was a MTRON MSD-SATA3035 with a 64 GB capacity and the other a SAMSUNG MMCQE28G8MUP-0VA providing space for 128 GB of data. In this system two graphics cards of type NVIDIA GeForce 285 GTX (MSI) with 1.7 GB VRAM and 240 streaming processors each running at 1.2 GHz were used. The system RAM amounts to 12 GB and was upgraded to 24 GB during the evaluation.

The operating system in use on the 32-bit system is SUSE 11 with CUDA 2.1 SDK and the NVIDIA driver version 177.13. On the 64-bit system Ubuntu 9.04 with CUDA 2.2 SDK and the NVIDIA driver version 195.36.24 is installed.

2.3.1 Solid State Disks

By the time this work began the solid state disks started to appear on the market and the differences in performance between the manufacturers were significantly high. The first used SSD was a HAMA 3,5” SATA solid state device with 32 GB storage capacity.

Table 2.3: Comparison of the used external storage devices in this work, given by the manufacturers.

Solid State Drives		
Manufacturer	MTRON	SAMSUNG
Model	MSD-SATA3035	MMCQE28G8MUP-0VA
Storage Capacity	64 GB	128 GB
Sequential access speed	81 MB/s	90 MB/s
Hard Disk Drives		
Manufacturer	SAMSUNG	WDC
Model	HD103UJ	WD5002ABYS-02B1B0
Storage Capacity	$1,000$ GB	500 GB
Sequential access speed	100 MB/s	100 MB/s

With further development of the hardware new devices were added, what enabled the possibility to build up a RAID with different devices and test grouped configurations. Table 2.3 compares different properties of the hardware. Unfortunately the HAMA disk broke during intensive evaluation and is not accessible for further investigation, so it was replaced and the experiments repeated.

Although Ajwani *et al.* have done exhaustive experiments to compare an SSD to magnetic media this work utilizes arrays of combinations including several SSDs and HDDs which makes it necessary to run a detailed evaluation.

This evaluation is done using iozone[7], a standard implementation for external media analysis. When testing external devices in a system with the Linux operating system one has to consider that the operating system will cache the data written to the disk in the internal memory to speed up the access. A solution to circumvent this problem is a size of a testing file which exceeds the internal memory. According to the system memory, the file size of a test file needs to be at least 32 GB to circumvent the internal cache. Provided this, a test was run to measure the differences in random access speed of the different devices.

External algorithms often require large amounts of storage. This can be achieved either by utilizing one large storage device or by combining several devices to a RAID array. In the used software RAID the kernel of the system distributes the data to several available devices. To find out whether an algorithm profits from an array of solid state drives several devices where evaluated in such a setting.

Sequential Access

Plots 2.6 and 2.7 show the access speed while writing and reading a file sequentially compared to the used data block size. The plots identify the hard disk drives to be clearly superior to the solid state devices. While the writing speed comparison points out a nearly double writing speed for the HDD the reading speed difference is less

[7] http://www.iozone.com

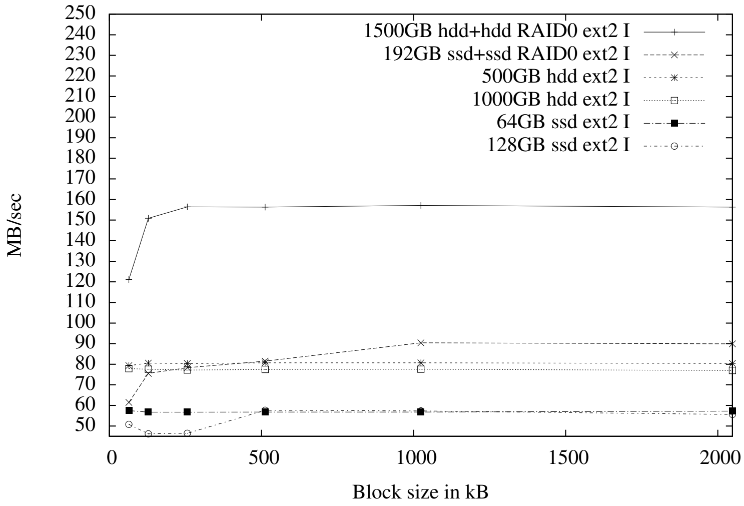


Figure 2.6: Sequential writing speed while writing a 32 GB file with different block sizes.

significant. The RAID array experiment exemplifies the disadvantages of the SSDs in writing even more, here adding a second device increased the speed only by 50%.

Figure 2.7 visualizes the disadvantage of the solid state media in reading small even sequentially aligned blocks of data. Since the media is not able to align the data for a fast adjacent access, the reading speed decreases with a decreasing block size, while the hard disk drive profits from its cache and continues to read the same block and achieves its maximum reading speed even for small block sizes.

This evaluation forces the algorithms to be developed for SSD usage to avoid writing, but when its inevitable it can be done even in small blocks. While in reading the block size should be as large as possible to achieve a maximum data throughput.

Figures 2.9 and 2.8 describe the performance in a random access situation depending on the size of the accessed block. Here the difference in writing and reading performance is even more significant. Due to the fact that each writing access imposes a copying of a block in the solid state device random writing of small blocks of data is a highly inefficient task. But also the hard disk devices decrease drastically in throughput due to the head movement necessary to write the data. The second solid state device can even cope with the speed of the hard disk drive in random writing speed.

The best use case for a solid state device is demonstrated in Figure 2.9 which points out the random reading performance of the compared devices. While the hard disk drives suffer from the high number of head movements when reading small blocks at random positions, the solid state drives reach the peak speed at even for small blocks and deliver the smallest chunks of data with a reading speed comparable to the maximal speed of the HDDs. One additional point should be mentioned when looking at the random reading speed of combined solid state devices, while combining two HDDs

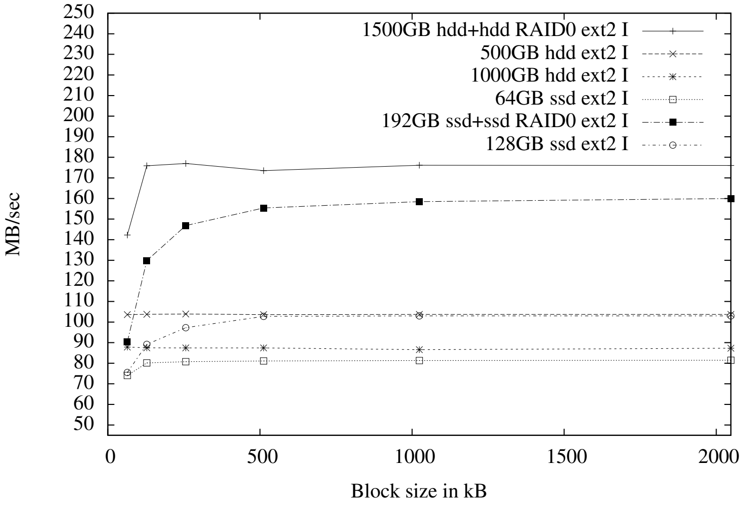


Figure 2.7: Sequential reading speed while reading a 32 GB file with different block sizes.

results in a mandatory increased random access speed the combination of 2 solid state drives can nearly reach a throughput being equal to a sum of both drives.

Algorithms with a high demand of random reading access to data can profit from the novel storage media even more if the devices are combined and arranged in an array.

2.3.2 Graphics Cards

To measure the performance of a graphics card two sets of experiments were conducted using the cards displayed in Table 2.4. The GTX 480 card was added to the system at the end of this work due to the fact of not being available when the work started.

The GTX 200 chip on the GTX 285 cards contains 10 texture processing clusters (TPC). Each TPC consists of 3 streaming multiprocessors (SM) and each SM includes 8 streaming processors (SPs) and 1 double precision unit. In total, it has 240 SPs executing the threads in parallel. Maximum block size for this GPU is 512. Given a grid, the TPCs divide the blocks on its SMs, and each SM controls at most $1,024$ threads, which are run on the 8 SPs.

The GTX 480 card uses a GTX 400 chip the next generation of cards, the *Fermi* architecture where a streaming multiprocessor comprises 32 streaming processors, each one with a double precision unit. The maximum block size is $1,024$ and the shared memory was increased to 48 kb per multiprocessor.

Due to the different clock rates of the systems the theoretical speedup factor is $240 * 1.2GHz/2GHz = 144$ and $(240 * 1.2GHz)/(2.67GHz * 4) = 26.966$[8] or even

[8] Taken into account that the CPU can execute up to four operations with one clock pulse.

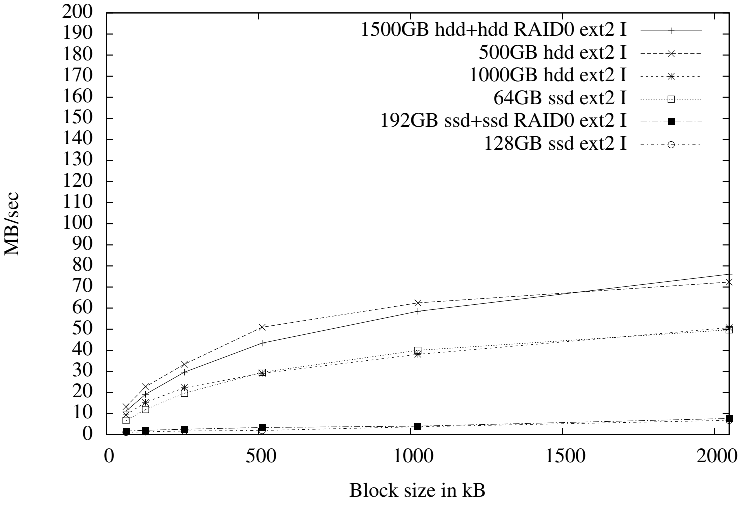


Figure 2.8: Comparison of random writing throughput.

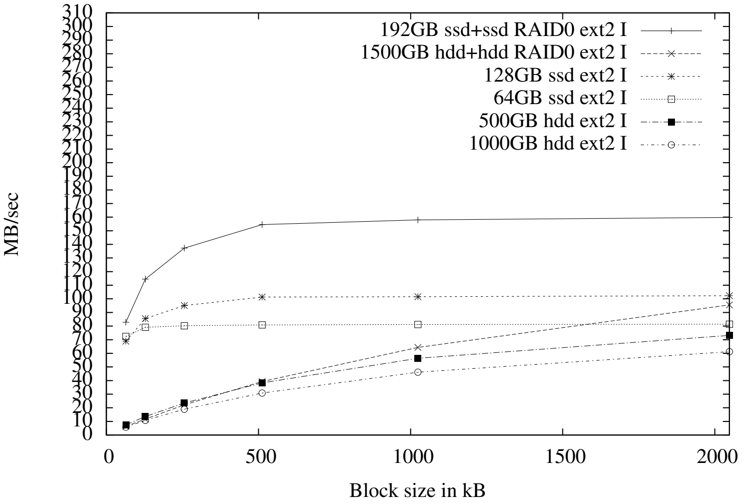


Figure 2.9: Comparison of random reading throughput.

Table 2.4: Comparison of the used graphics cards.

Type Used in System	GeForce GTX 285 32-bit	GeForce GTX 480 64-bit
Cores	240	480
GPU frequency	648	700
SP frequency	1,476	1,401
VRAM size	1,242 MB	1,536 MB
Memory speed	159 GB/s	177 GB/s

$(480 * 1.4Hz)/(2.67GHz * 4) = 62.92$ however, due to memory latency on both sides such a factor is not to be expected in complex algorithms.

The hardware systems were evaluated prior to the development of the algorithms by running basic experiments. In this section the transfer speed between the GPU and the host, the latency induced by starting the kernels and compared a GPU implementation to a CPU implementation on a highly parallel task is tested. All tasks performed on the 64-bit system evaluate the utilization of one GPU, since the GPUs are identical the performance is the same.

Figure 2.10 shows the transferring speed of data to the GPU. In the experiment a data block with a fixed size was copied 1.000 times to the GPU and the transferring speed computed. It can clearly be seen that the speed is dropping for blocks smaller than 5MB, and achieves its maximum of about 5.5 GB/s on the 64-bit and 1.5 GB/s on the 32-bit system when copying blocks larger than 5MB. Since the data transferring speed mostly relies on the bus no significant difference between the two cards in the 64-bit system are visible.

To test the latency introduced by starting the kernels three experiments were performed on each system. In each experiment a different number of blocks with 512 threads each was started 1,000 times. Figure 2.11 shows a decreasing number of kernel executions with a rising number of threads executing the *empty kernel*, a nearly empty function only assigning a value to a local variable. While the amount of threads increases linearly, by $10,000$ blocks the execution time decreases not linearly indicating that the GPU is able to schedule the work better over its processors with a higher number of threads. This figure also points out the drastic decrease in performance when accessing the global memory, a task that has to be performed in nearly all GPU algorithms. Using the *memory kernel*, a function that accesses the global memory to assign a value to a local variable, the performance drops by a factor of nearly 5 for both systems with the GTX 280 card. In contrast the performance of the GTX 480 card, optimized for memory access, stays nearly the same when the number of threads is sufficient. An interesting fact revealed by this experiment is that the system does not have much impact on the execution speed of threads when memory access is included, both lines of the *memory kernel* with the GTX 280 card nearly cover each other.

The impact of accessing the memory randomly is shown in Figure 2.12. Here a function is used which assigns values to a local variable from the global memory being

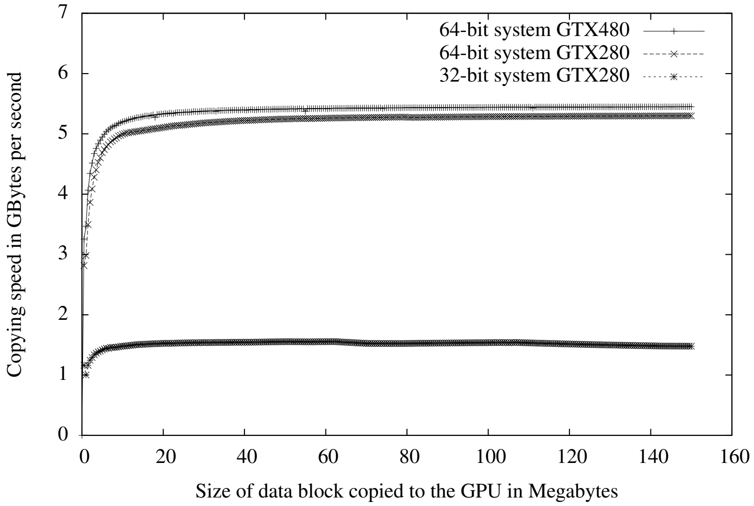


Figure 2.10: Transfer speed for copying data from host to the GPU tested for different sizes of data blocks.

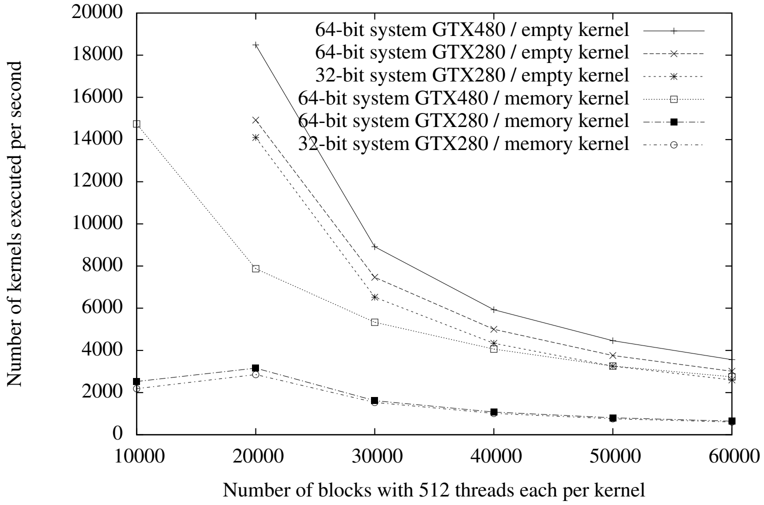


Figure 2.11: Kernels executed per second compared to kernel size and memory access, divided by system type. *empty kernel* denotes an empty kernel function while *memory kernel* identifies a kernel with one read access to global memory.

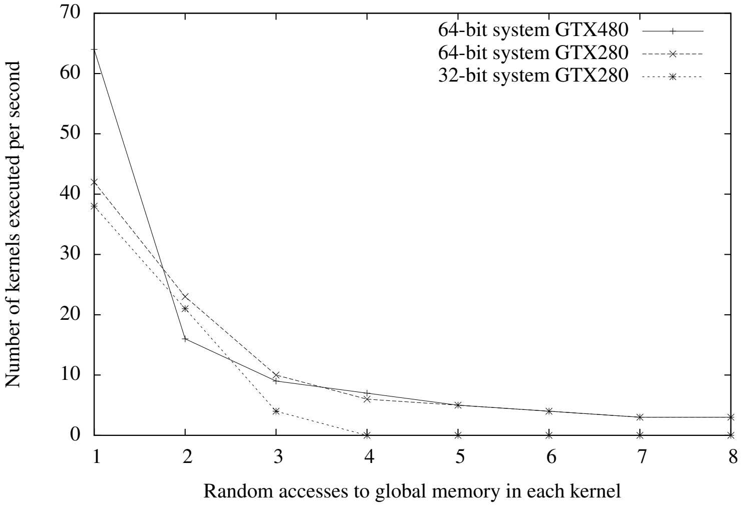


Figure 2.12: Kernels executed per second compared to number of random memory accesses.

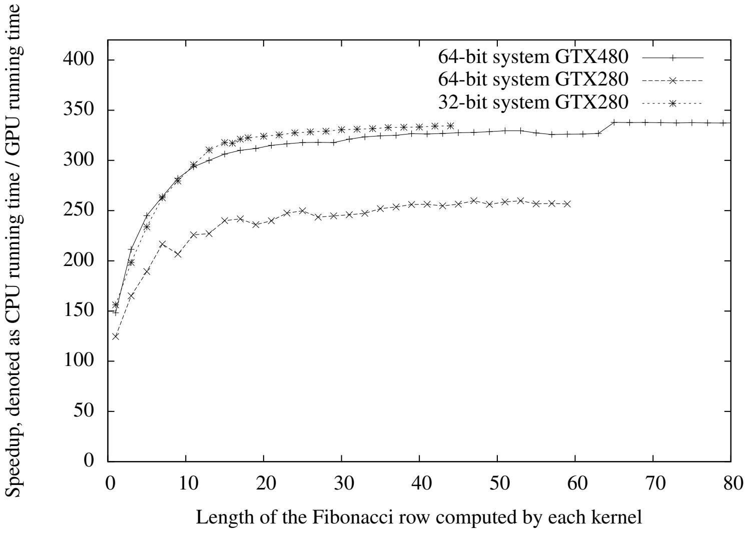


Figure 2.13: Maximal speedup achieved on the highly parallel task of computing a large amount of Fibonacci numbers.

scaled by the number of assignments. While in the function visualized in Figure 2.11 the memory access is aligned (thread i accesses cell[i] and thread $i+1$ cell[$i+1$]) this is not the case for Figure 2.12. The impact can be clearly seen as only 4 or 5 random reads suffice to slow the GPU down to execute just 3 iterations with 60.000 groups of 512 threads per second. Compared to aligned memory access the impact is even more obvious, while the GTX 480 is able to execute nearly 3000 iterations with 60.000 groups per second using aligned access, the iteration number drops to 65 when the access is unaligned.

An experimental setup is needed for testing the maximal speedup. Such a setup should be highly parallel so all the threads are performing nearly the same task. Additionally, such a task should be easy to scale. I choose to compute a large number of Fibonacci numbers using a function with 3 local variables and scaling the computation by the length of the Fibonacci row. Being computed, the number is written to the global memory and copied to the host to check its correctness. Figure 2.13 shows an impressive speedup of nearly 350 for the largest instances on both systems. Taking into account that the executed kernel has only one global memory access which is writing the computed number into its memory cell, and that all threads perform exactly the same task, a speedup of this size is not to be expected in realistic algorithms. The factor is much larger than the expected theoretical speedup, showing that the number of the processors on the GPU even adjusted with the clock rate cannot be used to determine the real speedup. The GTX280 card, solving the task in a comparable speed on both systems is able to achieve a higher speedup on the 32-bit system, visualized in the top position of the data plot, due to the slower 32-bit CPU. This experiment also reveals that doubling the number of cores on the GPU results not in a doubled speed, even for such highly parallel tasks.

Part I

Breadth-First Search utilizing Novel Hardware

Chapter 3

Prerequisites for GPU and SSD Utilization

Explicit graph algorithms utilizing the graphics processing unit (with a state space residing in RAM or on disk) were presented e. g., by Harish and Narayanan (2007). In this work, however, state space graphs are generated implicitly, by the application of transitions to states, starting with some initial state. Additionally, considering the fundamental difference in the architectures of the processing units solutions developed for multi-core Model Checking (Holzmann and Bosnacki, 2007) hardly transfer to ones for the GPU.

While analyzing the state space generation according to the description of an implicitly given graph the process will be divided and partitions, suitable to be ported to the graphics card or supported by external storage identified. Selecting appropriate subpart of a search algorithm implies braking it up and classify the partitions. Each algorithm can be seen as an ordered sequence of *tasks* to recieve the desired result. When described in pseudocode each line can be assumed as a task with a *function* grouping together subtasks. This function is also a task being a subtask of the whole algorithm. The tasks of an algorithm are classified here into *structured memory* tasks, grouping together the set of tasks which access only predefined locations in memory, and *unstructured memory* tasks, accessing either an unpredictable amount or unpredictable regions in memory. This classification is necessary to efficiently distribute the algorithm.

Structure of the chapter: This chapter will introduce basic analyzing techniques to identify parts in existing algorithms suitable for execution on the graphics card. The algorithm is examined from two points of view. On the one side the efficient distribution of performed work is considered on the other side a logical placement of information is proposed.

3.1 Work Distribution

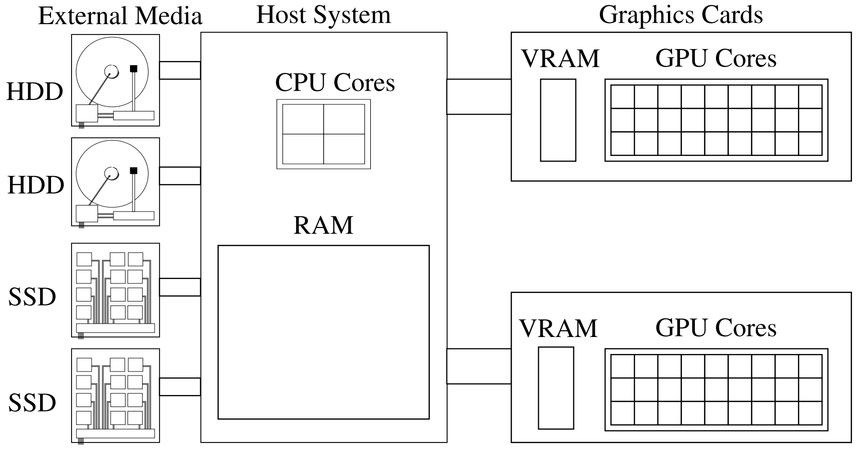


Figure 3.1: Visualization of the computing system used in this work. The host system, including the internal memory and the Central Processing Unit is the connecting part between the graphics cards and the external media. Currently there is a limit of 4 graphics cards in a system and a limit of maximal 128 external media devices.

The system considered in this thesis is composed of two computation areas connected to a number of storage devices. As visualized in Figure 3.1 the GPU computation processor, constructed of a high number of SIMD GPU cores, and the CPU which combines a number of independent CPU cores, are connected to each other via a bus. The cores do not only differ in data processing but also in memory access and processing speed. For an efficient work distribution the search algorithm is divided into tasks classified by their memory requirements. The classification in memory requirements is motivated by the fact that a search algorithm basically applies two operations to each state:

1. Generate all successors of a parent.
2. Check for each generated successor if it was seen before.

The naive approach, also presented in Algorithm 1.8 on Page 19, is doomed to failure here due to a high communication demand to distribute the states and manage a common $Closed$ and even $Open$ list. Evaluating the suitable tasks for the GPU and the CPU memory is done by dividing the algorithm and analyzing the memory requirements of each subtask performed by the algorithm.

To identify functions which should be ported to be executed on the GPU a classification rule is unavoidable. Such a rule has to take into account the dependency between the parallel tasks and the memory requirements of them. The term *output memory* of a function or task is used to denote the amount of memory necessary to store the result of a function, while the *evaluation memory* is the amount to evaluate the function.

3.1.1 Independent Limited Memory Tasks

Limited memory tasks denote tasks which access memory in a system of ordering and amount predictable before the execution. A limited memory task has a constant running time and if more of them can be performed simultaneously without a form of communication or synchronization they are *independent*.

Definition 24 (Independent Limited Memory Task) *An* independent limited memory task *is a part of an algorithm, e. g., a function, which satisfies two conditions.*

1. *The amount of used evaluation and output memory is known prior to running the task.*
2. *When running such tasks in parallel no communication is needed, i. e., the functions are independent.*

Independent limited memory tasks can be executed efficiently in parallel on the GPU, given the size of the evaluation memory and output memory is adequate to the card. The memory condition is a necessity to decide if the task is executable on the given card due to different amounts of available memory on different cards. Since the memory requirements are known prior to the execution there is no problem to decide whether the requested size fits on the particular GPU. Synchronization between processors in the GPU is not supported, and a communication over the slow global memory should be avoided in GPU suitable tasks. Now, having identified independent limited memory tasks as *GPU friendly* the algorithm can be partitioned and suitable tasks identified.

Generating Successors as an Independent Structured Memory Task

When generating successors for distinct parents the communication condition is satisfied. The problem is the amount of output memory utilized by this operation. It depends on the number of successors to generate.

If the upper bound for the number of successors of a state s is $max(s)$ and the amount of memory to store a state is denoted by $|s|$ successor generation is an independent memory task if $max(s) * |s|$ is reserved as output memory. This function would satisfy both conditions but is not practicable since the upper bound for the number of successors is not always known, and if so it would be an inefficient waste of memory if most parents have fewer successors than $max(s)$.

A second strategy is to divide the task into two subtasks:

1. Determine the number of successors.
2. Generate each successor independently.

Counting the number of successors for a state is an independent limited memory task due to the fact that it needs only an integer value as the output length. It can be done by processing only the preconditions or by generating all successors and discarding them immediately after returning the result. This process needs $|s|$ and the transition description as evaluation memory and is completely communication free between different states.

Generating the successors is also an independent structured memory task when implemented in the right way. One way is to construct pairs (s, t) of a parent and information which transition to enable. This way the parent is transformed into its successor and no additional evaluation memory beside the transition description is required. Communication is also avoided.

Having examined the algorithm for GPU friendly tasks the remaining functions can be classified as GPU unfriendly or as *unlimited memory tasks*. The term *dependent* is avoided since this tasks possibly are independent but are disqualified due to their memory requirements.

3.1.2 Unlimited Memory Tasks

Unlimited memory tasks should be maintained on the CPU side of the system because of two main facts. On the one hand the built-in structures of the processors can be used to implement an efficient communication, on the other hand the larger amount of RAM supports dynamic memory allocation. Since the search includes several unlimited memory tasks the CPU will control the whole process.

A decision rule for identifying independent limited memory tasks is a divide and conquer strategy (Brassard and Bratley, 1996). Divide the algorithm in logical tasks and check if they are limited in memory and independent. Otherwise proceed by dividing one of the tasks further. Of course a point exists where dividing does not make sense any more, i. e., when the task to divide is not parallelizable.

Checking for Duplicates as an Unlimited Memory Task

Every duplicate detection strategy using a $Closed$ list requests potentially access to the whole $Closed$ list. While the memory condition can be satisfied in the duplicate detection, by considering the evaluation memory to be the complete $Closed$ list and the output memory being just the information *expanded* or *unexpanded* the process still requires communication, thus it cannot satisfy the independence condition.

Consider two duplicates $s_0 = s_1$ to be checked using hash based duplicate detection in a sequential manner by two system reads. When thread t_0 is checking for existence of s_0 in $Closed$ there are two possible situations, s_0 is new so it has to be inserted into the $Closed$ list or s_0 already exists in $Closed$ and can be discarded. s_1 will be discarded in both situations, when s_0 is already visited s_1 is also, and if s_0 is inserted, thread t_1 will find it in the list and classify s_1 as seen. So the order of execution decides whether s_0 or s_1 will be inserted what is indifferent since they are duplicates. But when inserting in parallel, both threads would identify the non-existence of the corresponding state and try to insert it, unless some point of locking[1], at least at the table entry level, is done.

However, in a hash based approach this task can be divided further into:

1. Compute hash value for the state.

2. Check the state for existence in $Closed$.

[1] One thread occupies a memory region exclusively.

This partitioning extracts the hash value computation as an independent limited memory task which needs the evaluation memory to compute and the output memory to store the hash value. No communication is necessary when computing the hash values for a number of states, the first subtask. Looking up in the hash table still enforces communication, for the same reason as the whole duplicate detection.

A further unlimited memory task is to maintain the $Open$ list. Here the function has to select states to expand and has to consider an unpredictable amount of states in $Open$.

3.2 Information Distribution

Distributing all the maintained information in the system expects an analysis and a distinction of different information classes. Usually in algorithms two classes of information can be distinguished, constant and dynamic information. In this work the definition of the graph is considered as the constant and the description of a state as dynamic information. Although a generated state will never change during the search, every algorithm will expand each state only once and replace its position in memory with the following one, while the graph definition will remain constant over the whole search.

Basically the storage of information is given by the design of the algorithm. The information should be stored accessible for the task. The next two sections will classify the information which arises during a graph search and classify them in two groups.

3.2.1 Constant Information

Constant information is data which never changes during the search process. In graph search the transition rules are classified as constant information since they define the graph structure which remains constant over the search. Of course the algorithm implementation is also constant information, but is not considered in the analysis due to the fact that the operating system of the computer itself maintains it.

The state description could also be assumed as constant information since it never changes once a state is generated. To avoid such a classification this work considers only information structures which are known prior to the search and the state descriptions are not.

Constant information should be stored very efficiently since it is usually accessed frequently during the search process. In the case of the transition rules the information is accessed once for every node in the graph. The distance to the processor needing this information should be as short as possible, an optimal case is a representation which fits into the cache of such a processor and is accessed with nearly no latency. In case this information is stored on the GPU all pointers[2] are to be avoided, so the representation has to be flat and, if possible, sequentially accessed to enable an efficient access.

All other information in the search process is dynamic, either because it arises during the search or because it is extended or changed.

[2] A *pointer* is a variable type whose value refers another value stored in the computer memory.

3.2.2 Dynamic Information

A graph search utilizes three types of storage. The $Open$ list, being extended, the $Closed$ list, being either extended or changed, and the state representation, being generated, changed or even discarded during the search.

Since the state representation is system dependent and should be developed for each system independently there is only one strategy to mention here, i. e., compression. The number of states is usually not known before a search so an efficient storage using compression methods is an advance for it. However, if the compression is to hard to compute it may slow down the process, but still a search may only succeed on a given system before filling its memory due to an efficient compression.

The remaining structures i. e., the $Open$ list and the $Closed$ list, have basically only two levels of memory to reside, the RAM and the external memory. Graphics card memory is not suitable here because of its size compared to the system memory and lack of random read support.

The dynamic information can be classified further by counting the number of accesses to its stored states. While the $Open$ list entries are accessed basically once for an expansion, the entries in the $Closed$ list are accessed an arbitrary number of times depending on the structure of the graph.

If states in the $Open$ list can be grouped together and adjacent states are accessed in a block-wise manner, such a structure is perfect to reside on the external storage. Since this work mainly concentrates on BFS search the $Open$ list will be analyzed in detail in the following, also motivating to store it externally.

$Open$ list on external storage

The $Open$ list in a Breadth-First search is per definition a first in first out (FIFO) structure, so a state is accessed after all states being inserted before it are removed. Implementing such a structure is fairly simple by adding new elements to the end of a list and removing them, when needed, at the front. When space is not a constraint, like on external storage, or the elements are needed for further processing one can abandon the deletion and mark them as read. With such an approach the list's size will continually increase.

Such a list can be easily ported to the external block device by utilizing a memory portion as a buffer before writing and while reading the elements. When new elements are created they are stored in the buffer, which is appended to the end of the file once it gets filled. Reading elements from disk is realized in the backwards manner. When new states are needed the buffer is filled with states from the beginning of the file and the position of the read pointer is memorized marking all states before it as used. Figure 3.2 exemplifies this setting. Another positive aspect of such a storage, beside the low internal memory requirements, is the persistent storage of all elements after the search. Utilizing more than one external block device can be realized by an increased buffer and a RAID storage array, or by multiple buffers, one for each medium, and parallel writing and reading.

There is another important argument to store the $Open$ list on external media, i. e., the element size. System representations stored in the $Open$ list have to be compressed

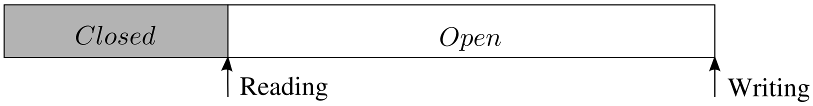


Figure 3.2: A single file represents the external $Open$ list structure in BFS. The reading pointer is used to divide expanded elements from unexpanded.

lossless, meaning that the representation has to be reconstructed from the compressed element. Since the elements in $Open$ have to be expanded all information available in the parent state has to be available again when the successor is generated.

Having clearly motivated to store the $Open$ list on external media the storage position of the $Closed$ list is much more dependent on the searching environment.

Where to store $Closed$?

A definitive answer, like in the previous section, cannot be given to this question, since it clearly depends on the strategy used for the $Closed$ list. The next few paragraphs will motivate several strategies to maintain $Closed$ and propose the storage of it. A complete list of all strategies for maintaining the $Closed$ structure would clearly exceed the scope of this work, so this section restricts to the strategies used in this work.

Sorting based $Closed$ In sorting based $Closed$ lists the elements are stored externally on block devices to utilize a minimum of internal memory (Korf and Schultze, 2005). Here the elements are sorted in an internal memory buffer prior to storing them on the external media. While this approach is very effective in avoiding internal memory usage it is also very ineffective in evaluation time, since every generated element has to be sorted prior to be stored or abandoned. Especially when the graph includes many duplicates this strategy will slow down the searching process extremely.

Hash based $Closed$ This strategy is clearly seen on the opposite side of the spectrum. It is very effective in terms of evaluation speed in internal memory, but ineffective in memory usage. Hash based $Closed$ list strategies map the element to an index and look in a table at the index position if such an element already exists. Since two elements can be mapped to the same index due to collisions, the table entry has to contain a distinct representation of the inserted element which is then compared to the new one. The table is accessed at nearly random positions, defined by the hash function and the order of generated elements, thus a storage on external block devices is ineffective and should be avoided. However, in Model Checking SSD storage was analyzed and showed decent results in being suitable for hash based $Closed$ lists (Barnat *et al.*, 2008a). Here the time sacrificed for an access to the table can be recompensed with the nearly unlimited storage capacity.

Perfect hashing based $Closed$ Using perfect hashing for the $Closed$ list is a three step approach. In the first step a set of all elements which should be included in the hash table is generated, and in the second step a perfect hash function (PHF) is computed which assigns an unique index to each element. Now the PHF can be used to assign entries in a table to elements which should be checked. The advantage is the compression ratio of such an approach. While the PHF can be compressed to 1.4 bits per element the table can be only 1 bit per entry. Both structures imply random access to all its entries, thus should be stored in the internal memory. External memory is only used to store the elements prior to building the PHF. Nevertheless, an efficient storage of the PHF on solid state media has been applied during this work leaving only 1 bit per stored element in the internal memory and will be presented in Chapter 6.

All these strategies, and combinations of them, are evaluated in the reminder of this work and show different advantages and disadvantages depending on the problem and the graph structure. The following chapter will introduce a framework to partition a search algorithm and strategies for the successor generation and duplicate detection. The strategies are developed to be efficiently executable on the graphics card, or efficiently utilize solid state media.

Chapter 4

GPUSSD-BFS - A GPU and SSD supported Breadth-First Search

Building up on the basic parallel Algorithm 1.8 presented in Chapter 1.3.3 a set of strategies will be proposed to utilize novel hardware developments, e. g., SSD storage or GPU computation, to speed up searching scenarios. In this work a hierarchical memory structure of SRAM and VRAM located on the GPU and described in detail in Chapter 2 is assumed. Additional memory is available as RAM and as external memory on magnetic and solid state disks. After presenting strategies for an efficient successor generation in parallel it will give detailed description on duplicate detection techniques based on sorting and hashing. The chapter ends with the proposal of a framework denoted with GPUSSD-BFS which is the baseline for speeding up the search in a state space.

Structure of the chapter The chapter starts with the proposal of a basic framework to utilize GPUs and SSDs in a Breadth-First search. Having exemplified three stages performed in any BFS algorithm strategies are given to port the state generation to the GPU, and perform efficient duplicate detection on the SSDs. The chapter continues with a proposal to translate Boolean formulas, often used in pre- and postconditions, into a representation suitable for the GPU.

4.1 Basic Structure of the Algorithm

The intuition behind this approach is to dispatch a set of operations to the GPU. For each BFS-Layer, the search is divided into two main computational stages performed utilizing the graphics card.

Stage 1 Determine the number of successors for a set of states in parallel.

Stage 2 Generate all successors in parallel.

followed by a third stage utilizing external memory

Stage 3 Remove all duplicate states and store new ones externally.

The pseudo-codes display a fine-grained algorithm, separating the selection of the transitions from their application. For the sake of clarity, the transfer from hard disk to RAM (and back) for layers that do not fit in RAM is hidden in the set based representation, so is the transfer from RAM to VRAM. In all remaining algorithms the copying of data to the GPU is hidden in the functions *fillVRAM* which denote the transferring of elements given to the function until the VRAM is filled.

Algorithm 4.1: Basic GPU Parallel Search algorithm

```
   Input: $\hat{s} \in \mathcal{S}$ initial state, $T$ set of transitions
 1 $Open \leftarrow \hat{s}$ ;                                      {store $\hat{s}$ in $Open$ }
 2 $Closed \leftarrow \emptyset$ ;                                 {clear $Closed$ list}
 3 while $Open \neq \emptyset$ do                        {repeat until search terminates}
             Stage 1 - Generate sets of active transitions
 4     Active $\leftarrow \emptyset$ ;                                      {clear Active}
 5     while |Active| $\neq |Open|$ do           {until all frontier states are processed}
 6         fillVRAM($u \in Open$) ;                           {copy states to VRAM}
 7         Active $\leftarrow$ Active $\cup$ GPU-Kernel Determine Transitions() ;
                  Stage 2 - Generate sets of successors
 8     Successors $\leftarrow \emptyset$ ;                              {clear Successors}
 9     while Active $\neq \emptyset$ do                     {Until all transitions processed}
10         fillVRAM(Active $\cup \{s, \cdots, s\}$) {Copy Active $\cup \{s, \cdots, s\}$ to VRAM}
11         Active $\leftarrow$ Active $\cap$ VRAM ;              {remove transferred states}
12         Successors $\leftarrow$ Successors $\cup$ GPU-Kernel Generate Successors();
             Stage 3 - Remove duplicates and rebuild $Open$
13     $Open \leftarrow \emptyset$ ;                                    {prepare next layer}
14     Successors $\leftarrow$ Successors $\cap$ $Closed$ ;              {remove explored states}
15     $Closed \leftarrow Closed \cup$ Successors ;           {extend set of explored states}
16     $Open \leftarrow$ Successors; ;              {add new layer to the search frontier}
```

Algorithm 4.1 depicts a framework to utilize the GPU in state space searching. Two sets denoted as *Active* and *Successors* are used beside $Open$ and $Closed$ to maintain information about the states. After the initial state is stored in $Open$ to be expanded $Closed$ is cleared and the search begins with Stage 1, the examination of the transitions. Here a subset of the states in $Open$ is copied to the VRAM (line 5) until it gets filled and a kernel is started to determine active transitions. The information about the active transitions is stored in the *Active* set when retrieved from the VRAM.

Having determined the active transitions, the successors are generated in Stage 2 where the VRAM is filled with the information from *Active* and additionally the states to expand. The successors are retrieved from the VRAM and stored in an array denoted as *Successors*.

When all successors are generated the duplicate detection phase is invoked in the third stage. Firstly, $Open$ is cleared since all states in this set are expanded, then all visited states are removed from *Successors*. Finally, the remaining states are appended to the $Closed$ list and $Open$. Naturally line 14 and 13 can be implemented in one step like in the hash based duplicate detection.

An additional aspect to mention is the scalability of the approach. Not only each stage can be distributed to several GPUs if they are available, but it is also possible to execute all stages in parallel. Once the *Active* set is filled stage 2 can start on the second GPU to generate the successors which are checked by the CPU when generated.

The sets *Active* and *Successors* can be externalized e. g., to use the internal memory for $Closed$, since they are accessed sequentially when they are populated.

Having defined the framework of a GPU algorithm, strategies for the three stages will be discussed in the following sections.

4.2 Strategies for Successor Generation

Splitting up the generation of successors may be a serious challenge depending on the system description. While in selected scenarios, like in Game Solving, the check for an enabled transition is trivially done by checking a bit, in scenarios like Explicit State Model Checking a precondition can be an arbitrarily complex Boolean formula. In the first case it suffices to return the number of successor in *Active* and recheck the preconditions. In scenarios with preconditions given in a Boolean formula a rechecking has to be avoided. Selecting a generation strategy for a given problem depends on several input variables. The next two sections will present two groups of strategies which can be applied efficiently when utilizing the GPU for successor generation. The first strategy is based on counting and is suitable for scenarios where the test whether a transition can be enabled is done without computation by a single lookup. Classifying the transitions whether they can be enabled or not, the alternative strategy is rather suitable when the check requires a decent amount of time.

Regardless of the strategy chosen to return information about the successors, the first two stages can already be classified by means of expected GPU performance. The task to determine the enabled transitions is an efficient SIMD task, since all threads perform exactly the same work by traversing a list of applicable preconditions and checking them for being active. In contrast to this the second stage, the application of enabled transitions depends highly on the length of the postconditions.

4.2.1 Successor Counting

The first strategy mentioned here is meant to determine how much space will be needed for the successors of a state. The advantage of it is the short output memory amount. Assuming the maximal number of active transitions $|t|$ is known, the output memory amounts to $log(t)$. In scenarios where the state description is also short, be it due to an

Algorithm 4.2: *GPU-Kernel* Determine Transitions for Successor Counting

```
Input: $\{s_1, \ldots, s_k\}$ array of elements to examine, $T$ set of transitions, $d$
       dimension of the grid
Output: $\{a_1, \ldots, a_k\}$ number of successor for each state in $\{s_1, \ldots, s_k\}$
for each group $g$ do in parallel                  {partially distributed computation}
    for each thread $p : 0 \leq p < d$ do in parallel       {distributed computation}
        $e \leftarrow 0$ ;                                           {reset counter}
        forall the $t \in T$ do                             {check each transition}
            if $t$ is applicable in $s_{g*d+p}$ then            {evaluate transition}
                $e \leftarrow e + 1$ ;                          {increase counter}
        $a_{g*d+p} \leftarrow e$ ;                      {replace state with number}
return $\{a_1, \ldots, a_k\}$ ;                         {return active transitions}
```

effective compression or simply a small state, this strategy helps to reduce the memory usage on the GPU for the set *Active*.

Determining active transitions

Algorithm 4.2 counts the number of active transitions and returns it as an array of numbers. The VRAM is filled with states prior to the execution of the kernel. When started, each thread analyzes a state based on its group and thread ID for enabled transitions and counts them in a sequential loop. The counter is maintained internally to reduce memory access to global memory. Having determined the number of enabled transitions it is stored in the VRAM at the position the state was before.

Since the states which are examined will usually be longer then the output variable a naive implementation is to replace the state by the number of its active transitions. An adjacent storage of variables is avoided since threads could overwrite states which are not examined yet, due to the lack of synchronization. This naive implementation needs the states to be copied twice to the GPU, once for examination of the active transitions and once for the successor generation. With a simple modification to this strategy the number of copies between the system and the GPU can be reduced at the cost of parallelism. When transmitting the states to the GPU a region should be left out which can hold a portion of the *Active* set.

The number of parallel instances reduces to $v = |VRAM|/(|s| + |a|)$ which is the size of the VRAM divided by the length of a state and the space occupied by a value returning the number of successors. Two arrays are needed on the GPU

$$\{\{s_1, \cdots, s_v\}, \{\text{room for } \textit{Active}\}\},$$

but the *Active* set can be copied back as one block without gaps. While copying this set the first part of the state array remains on the GPU for successor generation in step two. The drawback of this strategy which seems superior to simply overwriting the elements is the memory management. While coalesced reading is supported, coalesced writing is not, so each thread, writing into *Active* has to wait until neighboring threads have finished writing. In practice the naive approach was superior to this one.

Runtime Complexity Each state is examined by one thread for active transitions. The run-time is determined by the number of transitions times the number of groups, as for all threads in a group the transitions are checked in parallel.

Algorithm 4.3: *GPU-Kernel* Generate Successors for Successor Counting

```
Input: $\{s_1, \dots, s_k\}$ array of states to expand, $T$ set of transitions,
       $\{a_1, \dots, a_k\}$ numbers of successors, $d$ dimension of the grid,
       $Succ$ Successor generation function
Output: Successors the set of successors
1  Successors $\leftarrow \emptyset$ ;                                   {clear set of successors}
2  for each group $g$ do in parallel                          {partially distributed computation}
3      for each thread $p : 0 \leq p < d$ do in parallel             {distributed computation}
4          $g \leftarrow 0$ ;                                 {Counter for generated successors}
5          while $g < (a_{g*d+p})$ do                               {generate all successors}
6              forall the $t \in T$ do                                {check each transition}
7                  if $t$ is applicable in $s_{g*d+p}$ then               {evaluate transition}
8                      Successors $\leftarrow$ Successors $\cup$ $Succ(s_{g*d+p}, t)$ ;
                       {generate successor}
9                      $g \leftarrow g + 1$ ;        {increase counter for generated successors}
10 return Successors ;
```

Generating successors

Enabling transitions when the number of successors is known is realized by checking all transitions again, until the given number of successors is generated. Algorithm 4.3 exemplifies such a function. Here each thread analyzes one state for enabled transitions, applies the postconditions to the parent and stores its successor. Each thread expects a pair of information containing the parent and the number of successors to generate.

Two representations can be applied to store the generated successors in the VRAM. In both cases transferring the array of successor numbers $\{a_1, a_2, \cdots, a_k\}$ can be omitted when the line 5 is removed from the algorithm. Space for the successors can be allocated starting with the parent, followed by empty place according to the number of successors to generate.

$$\{\{s_1, \cdots\}, \{s_2, \cdots\}, \cdots, \{s_k, \cdots\}\}$$

There are two major drawbacks hidden in this representation. Firstly, an additional GPU buffer is necessary in the internal memory to distribute the states to their positions and secondly coalesced reading is unlikely to be applied due to the large gaps between the states.

The alternative, omitting the additional buffer, is to allocate the necessary space for all successors in one block of the global memory.

$$\{\{s_1, s_2, \cdots, s_k\}, \cdots\}$$

This strategy should be preferred since states located in $Open$ can be directly submitted to the GPU in one block and neighboring states are accessed simultaneously. However, since the size of a state is not limited coalescing reading adjacent elements is not guaranteed.

Both representations require a conversion of the successor numbers $\{a_1, \cdots, a_k\}$ to be usable by the threads of the GPU. Since each thread requires the position where to read the parent (in the first representation) or where to store the successors (in the second representation) a position array P contains the summed up number of all successors being generated until a particular index in the number of successors set.

$$p_i \in P = \sum_{j=0}^{i} a_j \text{ for the first strategy}$$

$$p_i \in P = \sum_{j=0}^{i} a_j - 1 \text{ for the second strategy}$$

Having generated the position array P each thread can immediately determine where to store the successors generated like described in Algorithm 4.3. It is efficient to copy P as one data block, since alternative implementations would imply a reorganization of the states on the CPU side.

Runtime Complexity Each thread generates all successors of a state to be expanded. The run-time is determined by the maximal number of successors times the number of groups.

4.2.2 Successor Pointing

When the check whether a transitions is enabled is complex it is inefficient to do it twice. Here a strategy is useful which propagates the information which transitions are active to the generation step. The challenge is to find a specification that does not exceed the size of a state and has a fixed length, independent form the number of active transitions.

Using a bit vector as a Bloom filter (Bloom, 1970) has shown the best results in terms of compression, each transition is compressed to a single bit, and the number of bits is constant over the search and corresponds to the number of available transitions.

Determining active transitions

The pseudo-codes for checking enabledness (Algorithm 4.4) and generating the successors (Algorithm 4.5) reflect that each processing core selects its share based on its group and thread ID just like in the pointing strategy.

Algorithm 4.4: *GPU-Kernel* Determine Transitions for Successor Pointing

```
Input: $\{s_1, \ldots, s_k\}$ array of states to examine, $T$ set of transitions, $d$ dimension
       of the grid
Output: $\{t_1, \ldots, t_k\}$ set of transition sets
for each group $g$ do in parallel                    {partially distributed computation}
    for each thread $p$ do in parallel                       {distributed computation}
        $B \leftarrow$ (false, ..., false) ;                   {clear applicable bit vector}
        forall the $t \in T$ do                                      {check transition}
            if $t$ is applicable in $s_{g*d+p}$ then                   {evaluate transition}
                $B[t] \leftarrow$ true ;          {check enabledness and set according bit}
        $t_{g \cdot sizeof(g)+p} \leftarrow B$ ;                       {overwrite selected state}
return $\{t_1, \ldots, t_k\}$ ;                        {return overwritten states to CPU}
```

In this strategy every thread allocates a bit vector B with a number of bits which corresponds to the upper bound of enabled transitions. Since each vector B can be significantly longer then a number representation a dedicated space in the global memory to returning the vectors is not to be efficient, leaving the replacement of states as the way to go.

There are two requirements for such a strategy to the system.

1. The upper bound of active transitions is known and constant for all states.

2. The transitions are ordered and indexed to be identified given the position of the corresponding enabled bit.

The given requirements are no restrictions in most of the systems since the maximum number of outgoing transitions is known from the definition of the implicit graph which also imposes an ordering on them. The reason for maintaining the vector B internally is to decrease the number of VRAM accesses to only one, when the analysis is done.

Runtime Complexity Each state is examined by one thread for active transitions. The run-time is determined by the number of transitions times the number of groups.

Generating successors

Preparation of data in the system is significantly different compared to the successor counting strategy. Since the information on how many transitions have to be applied can be interpreted from B, one could simply transfer the parents followed by the bit vectors and leave space for generated successors in the global memory.

$$\{\{s_1, s_2, \cdots, s_n\}, \{b_1, b_2, \cdots, b_n\}, \{\text{space for successors}\}\}$$

This storage strategy is not efficient on the GPU for a couple of reasons which partially are shared with the successor counting strategy e. g., different numbers of successors imposing different running times on threads, but apply also to this specific approach. Since the length of the bit vector is constant and independent of the number of enabled transitions, space is wasted if only a few bits are enabled. Additionally, each thread has to access two places of memory to be able to create the successors, namely the state-vector and the bit vector.

To circumvent this drawbacks a different storage strategy is necessary with a better support for coalesced reading and a more synchronized successor generation. Different numbers of successors in each state imply different lengths on loops generating the successors causing idle times for threads. A solution here is to choose a fixed number of generations for each thread. This number is one since each state transmitted to the GPU has at least one successor. The limitation to generate only one successor enables a different storing strategy.

$$\{\{s_1, t_1\}, \{s_1, t_2\}, \cdots, \{s_n, t_n\}\}$$

In this representation each thread reads a pair $\{s, t\}$ and applies the transition denoted with the id given in t to the state s. Since all threads apply only one transition the work is divided equally among them.

Algorithm 4.5: *GPU-Kernel* Generate Successors for Successor Pointing

```
Input: $\{\{s_1, t_1\}, \{s_1, t_2\}, \ldots, \{s_n, t_n\}\}$ array of pairs of state and transition to
       apply, $T$ set of transitions, $d$ dimension of the grid
Output: Successors the set of successors (explored nodes are overwritten)
for each group $g$ do
    for each thread $p$ do in parallel
        $Successors \leftarrow Successors \cup Succ(s_{g \cdot d+p}, t_{t_{g \cdot d+p}})$ ;     {add successor}
return $\{s_1, \ldots, s_k\}$ ;     {Feedback result to CPU}
```

An experimental evaluation identified the second strategy as superior. Therefore, this work proposes to replicate each state to be explored by the number of enabled transitions on the CPU. Moreover, attach the ID of the transition that is enabled together with each state. Then, move the array of states to the GPU and generate the successors in parallel overwriting the parent state.

Runtime Complexity Each state to be explored is overwritten with the result of applying the attached transition, which often results in small changes to the state vector. Finally, all states are copied back to RAM. The run-time is determined by the maximal length of an effect times the number of groups.

4.3 Strategies for Duplicate Detection

After having generated the successors on the GPU they are copied back into the internal or external memory and checked for duplicates. Since a set of successors is generated in the third stage parallel methods should be applied to find already expanded states. This section will propose three strategies to perform a duplicate detection utilizing the parallel processing power of either the GPU or the CPU and the storage capabilities of SSDs. When a low probability of removing unexplored states is acceptable, an incomplete duplicate detection can be used. Here different compression methods for both strategies are proposed.

The external memory sorting approach supports state space sizes up to the size of the external media, classified as unlimited in this work. However, sorting a huge number of elements is time costly, even when done on the GPU. In addition to the sorting time there is the latency which is caused by reading all previously generated states for each set generated by the GPU. The amount of time necessary in this step increases proportionally to the number of generated states.

4.3.1 Sorting Based Duplicate Detection

GPU-based sorting won the 2006 Indy PennySort category of the TeraSort competition (Govindaraju *et al.*, 2006), a sorting benchmark, testing performance for database operations. Since then, various GPU sorting algorithms have been proposed, including MP5[1] GPU BITONIC SORT (Batcher, 1968) and GPU QUICKSORT (Cederman and Tsigas, 2008). One of the best general GPU sorting algorithm was introduced by Leischner *et al.* (2010), the question is if this efforts can be ported to sorting long states.

Consequently an evaluation was performed to analyze both sorting algorithms in a state space search utilizing delayed duplicate detection. The initial results, implemented in an existing model checker DIVINE and documented in (Edelkamp and Sulewski, 2008c), were disappointing. Even after further refinements, the best improvement of existing GPU sorting technology achievable wrt. CPU QUICKSORT was about 20%. Lessons learned in this evaluation where that the size of the element has a crucial impact on the sorting speed, with a rising size of sorted elements the sorting speed decreased continually. Trying to sort a set of indexes also failed badly, as now the comparison exceeds the boundary of the SRAM, since elements have to be fetched from the VRAM and compared. For effective GPU-sorting the sorted elements should be as small as possible, and sorting has to stay local.

Assuming an efficient algorithm to sort this kind of objects exists the GPU can be used to seed up sorting in external breadth first search approaches. The strategy to remove duplicates is a two step approach:

Step 1. Sorting Sort the generated successors in a predefined order.

Step 2. Removing Remove neighboring duplicates.

[1] `courses.ece.uiuc.edu/ece498/al/mps/MP5-TopWinners/kaatz/MP5-parallel_sort.zip`

Now the set of successors is free of duplicates and has to be compared to previous sets stored on disk by scanning once through the *Open* file just like described in Section 1.3.2.

In the following a hybrid of sorting- and hash-based delayed duplicate detection is proposed, sorting buckets filled by applying a first level hash function. The hidden objective of this approach is that hashing in RAM allows distant data moves, while sorting only induces local changes and can be accelerated on the GPU.

Hash based partitioning

The BITONIC SORT approach which revealed to be superior to other sorting methods on the GPU consists of two phases. In the first one a block of threads is used to sort a subset of all elements that fits into the SRAM, then the sorted subsets are joint invoking intensive access to the VRAM. The crucial observation is, the first phase accesses the global memory only once for reading and after sorting once for writing, so it is fast compared to the second phase. Therefore, hash-based partitioning on the CPU was employed in order to distribute the elements into buckets of adequate size and use only the first phase of BITONIC SORT.

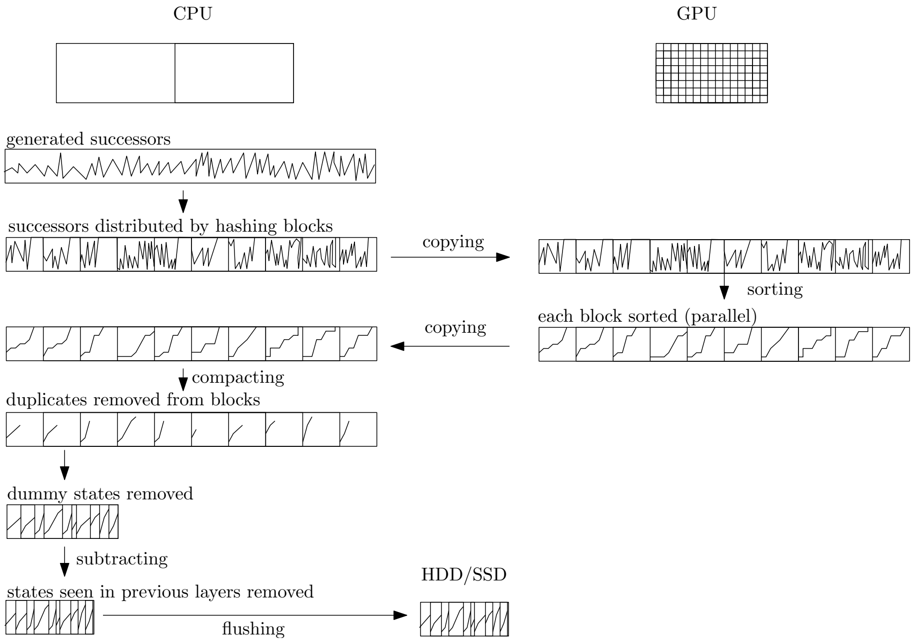

CPU
GPU
generated successors
successors distributed by hashing blocks
copying
sorting
each block sorted (parallel)
copying
compacting
duplicates removed from blocks
dummy states removed
subtracting
states seen in previous layers removed
HDD/SSD
flushing

Figure 4.1: Efficient sorting of large elements utilizing both the GPU and the CPU. The generated successors are distributed to blocks of equal size, using a hash function. After copying the buckets to the GPU, each bucket is sorted by a block of threads. Back in the internal memory all adjacent duplicates and states from previous layers removed.

The array to be sorted is scanned once as sketched in Figure 4.1. Using the hash function h and a distribution of the VRAM into p blocks, an element s is written to the bucket with index $h'(s) = h(s) \bmod p$. On the first overflow in one of the buckets, all remaining places in all buckets are set to a predefined illegal vector that realizes the largest possible value in the total ordering of elements. This hash-partitioned vector is copied to the graphics card and the buckets are sorted in parallel as indicated in Algorithm 4.6. A crucial observation is that the array is fully sorted wrt. to the extended comparison function operating on pairs $(h'(s), s)$. The sorted vector is copied back from VRAM to RAM, and the array is compacted by eliminating duplicates with another scan through the elements. Subtracting visited states is made possible by scanning all previous layers residing on disk. Finally, the current, duplicate-free BFS layer is flushed to disk and iterated.

Algorithm 4.6: *GPU-Kernel* Sort buckets in sorting based duplicate detection

```
Input: $\{H_1, \ldots, H_k\}$ (unsorted buckets)
Output: $\{H_1, \ldots, H_k\}$ (sorted buckets)
for each group $g$ do
    $SRAM \leftarrow SelectBucket(H_g)$ ;            {Copy bucket to SRAM}
    $H_g \leftarrow Sort(SRAM)$ ;                    {Sort and write bucket back}
return $\{H_1, \ldots, H_k\}$ ;                      {Feedback result to CPU}
```

As long as the files do not exceed the GPUs memory, the above exploration strategy is sufficient. If a BFS-Layer becomes too large to be sorted on the GPU, it is split into parts that fit in the VRAM. This yields additional state vector files to be subtracted to obtain a duplicate-free layer. For the case that subtraction becomes harder hash-partitioning can be exploited – inserting previous states into files partitioned by the same hash value – a technique inspired by hash-based duplicate detection (Korf and Schultze, 2005) and implemented in structured duplicate detection (Zhou and Hansen, 2004). Provided that the sorting order is first on the hash value and then on the state, after the concatenation of files (even if sorted separately) a total order on the sets is obtained. This implies that duplicate detection including subtraction can be restricted to states with matching hash values.

Incomplete Sorting Based Duplicate Detection Using State Compression

The shorter the state vector, the more elements fit into one bucket, and the better the expected speed-up on the GPU. For improving the sorting performance the state vectors are compressed to 64-bits (Stern and Dill, 1996); Roughly speaking, hash-compaction yields a Bloom filter (Bloom, 1970) variant for (single, double, or triple) Bitstate hashing (Holzmann, 1998). The objective is that false positives can arise during search. Two independent 32-bit hash functions h_1 and h_2 are chosen randomly from a set of universal hash functions. The state vector for s is compressed to $(h_1(s), h_2(s), a(s))$, where $a(s)$ is the index of the state vector residing in RAM that is needed for state exploration. The values $(h_1(s), h_2(s), a(s))$ are then sorted lexicographically on the GPU. The empirical observation is that with this 64-bit hash address no collision appeared even for very large state spaces [2]

Turning this approach into a complete duplicate detection is done by checking each state vector for equality with the address indexed by the pair $(a(s), a(s'))$ and $(h_1(s), h_2(s)) = (h_1(s'), h_2(s'))$ for $s = s'$. This check is costly due to the intensive random memory access, even if it can be obtained while scanning the data once.

Probability of false positive

For deriving an estimate on the probability of a false positive, assume a space of $n = 2^{30}$ states universally hashed to the $m = 2^{64}$ possible bit vectors of length 64. According to the birthday problem (Bloom, 1973), the probability of having no duplicates is $m!/(m^n(m-n)!)$. One known upper bound is $e^{-n(n-1)/2m}$, which in this case resolves to 0.9692, such that we have a chance of less than 96.92% to have no collision during the search. But how much less can this be? For a better confidence on our algorithm, a lower bound is needed. Lets have $m!/(m^n(m-n)!) \geq (1 - n/m)^n$. For this case this resolves to $(1-2^{-34})^{2^{30}} = (0.99999999994179233909)^{1073741824} = 0.9394$. Hence, a confidence of at least 93.94% that no collision arises is reached.

An alternative way of computing the error probability is as follows. There are $2^{30}(2^{30} - 1)/2$ pairs of states (x, y), where $x < y$. For a random hash function h, and for any such pair, the probability that $h(x) = h(y)$ is $1/2^{64}$. Therefore, the expected number of hash conflicts is $(2^{30}(2^{30}-1)/2)/2^{64} = (2^{60}-2^{30})/2^{65} = 1/2^5 - 1/2^{35} \leq 0.03126$, certifying that with a chance of more than a 99.68%, no false positive has been produced, while traversing the entire state space.

In contrast, single Bitstate hashing with an 8 GB-sized hash table results in an expected number of about $(2^{30}(2^{30} - 1)/2)/2^{36} \approx 2^{23}$ hash conflicts (see (Holzmann, 1998) for an analysis of single, double, and multi Bitstate hashing). Moreover, missing a duplicate harms, only if the missed state is the exclusive way to reach the target. In the search spaces examined in this evaluation the 64-bit compression did not miss any single element. If the above certified confidence is still too small, one can re-run the experiment with another set of hash functions, as in the Supertrace algorithm (Holzmann, 2004).

4.3.2 Parallel Hash Based Duplicate Detection

The advance of an unlimited space in sorting based duplicate detection is achieved by sacrificing time to access the data. Even the fastest external media devices can not cope with the speed of internal hash based duplicate detection due to two reasons: all states have to be compared to be sorted and all new states have to be stored on disk at a writing speed of about 100 MB/s compared to a writing speed of 17,000 MB/s in RAM. In contrast to the sorting approach a hash based one computes a hash value for every state and compares it to a small number of elements retrieved from the internal storage. So a hashing based approach should be preferred until the internal memory is exhausted.

[2] The approach reassembles ideas from Game Solving, where boards are often mapped to 64-bit vectors.

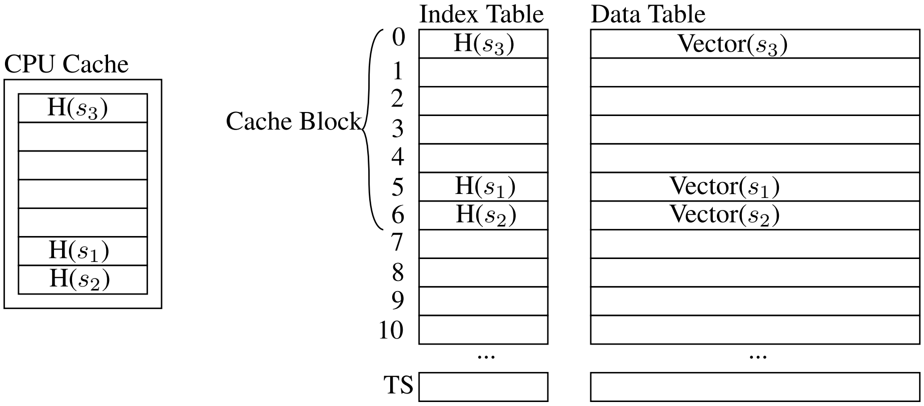


Figure 4.2: Table configuration in the Lock less Hashing approach. When a new state is to be inserted a *cache block* is transferred into the CPU cache and examined for existing hashes. To avoid memory accesses linear probing starts at a top of a *cache block* when the bottom is reached. Only when a cache block is filled a new one is fetched from the memory. On equal hash values the vector in the *data table* is compared to the inspected one.

The goal is to introduce an efficient shared state storage for immediate duplicate detection. Since in state space exploration, only storing and retrieving state vectors is necessary, the stored key is the state vector itself. The time efficiency of the lookup should scale with the number of cores utilized in parallel. Pointers and memory allocations are avoided. The implementation of the hash table uses open addressing tuned to use the cache more efficiently by small-sized linear probing. Lock-free algorithms guarantee system-wide progress in modern CPUs. They implement a compare-and-swap operation (CAS), so that always some thread can continue its work.

In this solution, based on the tables from Laarman *et al.* (2010) however, only statistical progress is guaranteed, explicit locks are avoided using CAS. This leads to a simpler implementation and no penalty in performance. Strictly speaking, the algorithm locks *in-situ* – it needs no additional variables for implementing the locking mechanism. CAS ensures atomic memory modification while at the same time preserving data consistency.

The problem with lock-free hashing is that it relies on low-level CAS operations with an upper limit on the data size that can be stored (one memory cell). In order to store states that usually exceed the size of a memory cell, two tables are needed: the *Index* and the *Data* table. Memorized hashes and the write status bit of the state is stored in the Index table. If h is the memorized hash, the possible values of the indexes are thus: $(-, \top)$ for being empty, $(h, \bot)$ for being blocked for writing and $(h, \top)$ for releasing the lock. In Index the locking mechanism is realized by CAS.

Parallel Bloom filter detection

Using a Bloom filter (Bloom, 1970) $Closed$ is compressed to only one bit per state addressed by the hash function $h(s)$. When a collision appears with $h(s) = h(s')$ and $s \neq s'$ the second state will be discarded even if it is unexpanded.

To decrease the possibility of discarding an unexpanded state the number of used hash functions can be increased and several bits addressed for each state. When using c hash functions a collision appears if $\{h_1(s), \ldots, h_c(s')\} = \{h_1(s'), \ldots, h_c(s')\}$ and $s \neq s'$. The state s will be also discarded if a configuration of c states $\{s_1, \ldots, s_c\}$ exist such that $\{h_1(s), \ldots, h_c(s')\} = \{h_1(s'_1), \ldots, h_c(s'_c)\}$.

Parallelizing is done by utilizing more then one thread for the check risking to visit some states more then once when two threads try to insert the same state at the same time. Even if just one hash function is used thread t_2 can check $h(s)$ before thread t_1 has set the corresponding bit to `true` hence, t_2 will classify s as new. Now both threads will try to insert s into $Open$.

In an environment where a perfect hash function is available a parallel Bloom filter (Bloom, 1970) can even be used for a complete search due to the absence of collisions. Since no collision can appear when computing the hash value for s using the hash function h, $h(s)$ identifies s uniquely. $Closed$ is then represented as a bit vector and a `true` bit at position i identifies the state $h(s) = i$ as expanded.

4.4 External State Space Exploration on the GPU

When the amount of RAM and the available GPU memory (e. g., when using multiple GPUs) amounts are nearly the same it is not efficient to store the *Active* and *Successors* sets internally. Here an extension to the framework will be presented to utilize RAM only as a buffer for the GPU. It is not possible to transfer data from the external storage to the GPU directly, so at least one buffer has to be maintained. However, the buffers are not exemplified in pseudo-code of Algorithm 4.7 since they can be seen as transparent.

The process of the exploration is divided in three stages as proposed by Algorithm 4.1. For each BFS-Layer the state space enumeration is divided into three computational stages (see Algorithm 4.7). In the first stage the elements in $Open$ are transferred into the GPU and the set of active transitions is stored in the file $Active_{ex}$.

In the second stage, sets of all possible successors are generated. For each enabled transition a pair, joining the transition ID and the explored state, is copied to the VRAM. Each state is replicated by the number of successors it generates in order to avoid memory to be allocated dynamically. Here it is efficient to have $Open_{ex}$ and $Active_{ex}$ on separate storing devices since a parallel read can be used.

The third stage removes all duplicates by using the hash based GPU sorting approach and hashing the successors to buckets, which are indexed by the hash value, and by sorting the buckets in the GPU. Adjacent duplicates are removed in a first scan, followed by scans to remove duplicates from previous layers.

Algorithm 4.7: GPU-BFS - Large-Scale Breadth-First search on the GPU

```
Input: $\hat{s} \in \mathcal{S}$ initial state, $T$ set of transitions
$g \leftarrow 0$ ;                                          {reset counter for the BFS-Layers}
$Open_{ex}[g] \leftarrow \hat{s}$ ;                                          {insert $\hat{s}$ into first file}
while $Open_{ex}[g] \neq \emptyset$ do                              {until an empty layer is found}
                                  {Stage 1 - Generate sets of enabled transitions}
    $Open_{ex}[g+1] \leftarrow \emptyset$ ;                                          {reset next file}
    while $|\text{Active}_{ex}| \neq |Open_{ex}|$ do        {until all frontier states are processed}
        fillVRAM($u \in Open_{ex}$) ;                                   {copy states to VRAM}
        $Active_{ex} \leftarrow Active_{ex} \cup$ GPU-Kernel Determine Transitions() ;
                                        {Stage 2 - Generate sets of successors}
    $Successors_{ex} \leftarrow \emptyset$ ;                                  {clear Successors file}
    while $\rceil\text{EndOfFile}(\text{Active}_{ex})$ do                  {Until all transitions processed}
        fillVRAM($Active_{ex} \cup \{s, \cdots, s\}$)
        {Copy $Active \cup \{s, \cdots, s\}$ to VRAM}
        $Successors_{ex} \leftarrow Successors_{ex} \cup$ GPU-Kernel Generate Successors();
                              {Stage 3 - Remove duplicates and rebuild $Open$ }
    for $s \in \text{Successors}_{ex}$ do
        $H[hash(s)] \leftarrow H[hash(s)] \cup \{s\}$ ;      {insert $s$ into bucket $H[hash(s)]$}
        if $|H[\text{hash}(s)]| = H[\text{hash}(s)].\max$ then                      {if bucket full}
            Sorted $\leftarrow$ GPU-Kernel sort buckets($H$) ;      {sort buckets on GPU}
            Compacted $\leftarrow$ ScanAndRemoveDuplicates(Sorted) ;
            DuplicateFree $\leftarrow$ SubtractDuplicates(Compacted, $Open_{ex}[0..g]$) ;
            $Open_{ex}[g+1] \leftarrow$ Merge($Open_{ex}[g+1]$, DuplicateFree) ;
            $H[0..m] \leftarrow \emptyset$ ;                                    {reset buckets}
    $g \leftarrow g + 1$ ;
return $Open_{ex}[0..g-1]$ ;
```

4.5 Efficient Flat Representation of Formulas

To check the transitions for enabledness, a representation of them has to be accessible by the GPU cores. While an object-oriented data structure – where each expression in a process is realized as an object linked to its substructures – might be a preferable representation of the graph definition for CPU access, such a representation would be less effective for GPU access.

As described in Section 2, the GPU's memory manager prefers sequential access to the data structures. Moreover, to use coalescing reading many threads have to access the same memory area in parallel. Hence, in order to speed up the access the pre- and postconditions should reside in the SRAM of each multi-processor. This way a fast randomized access can be granted, while the available space shrinks to at most SRAM size.

Since the GPU should not access RAM and pointer manipulation on the GPU is

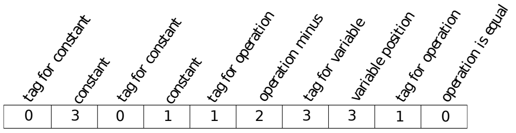


Figure 4.3: The precondition *my_place == 3-1* encoded in Reverse Polish Notation and stored in an integer vector.

limited, it is necessary to rewrite the condition labels to be evaluated. This description has to be efficient in terms of memory and evaluation time, due to the small size of the VRAM (compared to the computational power of the GPU). Furthermore, all transitions should be moved to the graphics card as one memory block to take advantage of fast block transfers on the express bus.

The challenge is to store the representation of the transitions efficiently. On the GPU, the *Reverse Polish Notation* (RPN)[3] (Burks *et al.*, 1954), i. e., a postfix representation of Boolean and arithmetic expressions, was identified as effective. It is used to represent all preconditions and postconditions of the model in one integer array. This array is partitioned into two parts, one for the preconditions, the other for the postconditions. A prefix assigns the conditions to its processes after creating the array. In addition to the preconditions, each transition indicates the successor state the process will reach after enabling the postconditions. *Tokens* are used to distinguish different elements of the Boolean formulas. Each entry consists of a pair (token,value) identifying the action to take. Consider the precondition `my\_place==3-1;` starting at position 8 of the array presented in Figure 7.2. It is translated to the RPN as an array of length 10 using tokens for constants, arithmetic operations and variables. Constant tokens, defining also the type of the constant, are followed by the value. Arithmetic tokens identify the following byte as an operator. One special token is the variable token, there is no need for distinction in arrays or variables, since variables are seen as arrays of length 1, so the token defines the type of the variable and is followed by the index to access it in the state. This yields a pointer-free, compact and flat representation of the transition conditions.

Evaluation complexity To evaluate a postfix representation of a boolean formula, one scan through its representation suffices. The maximal length of a formula times the number of groups thus determines the parallel running time, as for all threads in a group, the check for enabledness is executed concurrently.

4.6 Summary

In this chapter an algorithm was proposed to utilize the highly parallel graphics processing unit located on a graphics card and the corresponding memory to speed up state

[3] Originally invented by the Polish logican Jan Łukasiewicz in the 1920s.

space generation and searching. For an efficient duplicate detection and the *Open* list solid state devices are utilized due to their improved transferring speed.

The framework algorithm consists of three stages from which two use the GPU for generating the successors and the third one stores the generated states on external media to increase the available storage space. In the first stage a set of states is transferred to the GPU and tested for active transitions. In the second stage a set of transitions assigned to states is copied to the GPU and the successors are generated in parallel on the device. In the third stage a parallel duplicate elimination is performed and visited states are removed. This elimination can be either complete or not complete where a small probability of removing new states is given. For each stage several strategies were proposed to be used in the framework.

Determining active transitions is performed either in the successor counting strategy, here the output of the GPU-Kernel is the number of successors to generate or using the successor pointing strategy and identifying each successor uniquely by the index of a bit in a vector.

When just the number of the successors is known, the generating function uses it for allocating memory on the graphics card and generating the successors by repeating the precondition check. The successor pointing strategy trades space for a returning bit vector for a reduced work in the generation phase due to an avoided repetition of the precondition check.

Duplicate detection can be either delayed by utilizing a sorting based approach and external media or done semi immediately by utilizing lock less parallel hashing using internal memory. Both strategies can be restricted to reduce the completeness and increase storage capacity and lookup speed by using a compression in the first approach and a bloom filter in the hash-based duplicate detection.

Still, there are remaining obstacles in implementing a fully-fledged state space search on the GPU. First, the state size may vary during the verification. Fortunately, the analyzed domains provide upper bounds on the state vector size or induce the maximal size of the state vector once the initial state has been read. Another technical challenge is that the GPU kernel (though being C-like) does not exactly match the sources of existing searching implementations, such that all methods being called have to be ported to CUDA.

Part II

Explicit State Model Checking

Chapter 5

Introduction to Explicit State Model Checking

Developing software is an evolving process. Even for systems where specifications exist prior to the implementation each new version has to be checked whether it complies with the specification. Since validating the specification on the implementation of a prototype may not be possible a representation has to be generated and maintained. The representation is then verified to fulfill the properties given in the specification. After being checked the representation is converted into a programming language compiled for the given system. On the other hand an implementation can be analyzed to generate a representation and check it. Both approaches demand for high computation and storage capabilities not only for the transformation task, but also for the checking process.

A common approach to verify a system is *Model Checking* (Clarke *et al.*, 1999; Müller-Olm *et al.*, 1999) where a representation of the system, called *model* is created and checked against given properties. Checking is done by traversing an implicit graph defined by an initial state and transition definitions. The resulting graph can be arbitrarily large and modifications on the underlying model, e. g., while extending it to a new version, can increase the number of nodes in the graph even exponentially. To handle the large number of nodes and avoid the state space explosion problem, generated by a large branching factor, algorithms are needed that use the available hardware efficiently.

Structure of this chapter: In the remaining part of this chapter Model Checking will be defined in detail linking it to the state space search approach. The thesis example from the introduction will be formed as a model in the DIVINE description language DVE. The chapter closes by giving an overview on the related work.

5.1 Modeling of Concurrent Systems

While the definition of Model Checking is as follows:

Definition 25 (Model Checking) *Given a Model of a system $\mathcal{M}$ and a property specification ϕ, Model Checking verifies if ϕ is satisfied by the system, ($\mathcal{M} \models \phi$)*

several formalisms exist to model a concurrent system, e. g., communicating sequential processes (CSP) (Hoare, 1978), petri nets (Petri, 1962) and process algebras such as the calculus of communicating systems (CCS) (Milner, 1980). When defined, the model should represent the system as accurate as possible, since differences between the representation and the prototype to verify may lead to false positives, finding errors in the model not being present in the system, or to an overlooked error in the system. The notification used in this work is equivalent to the DVE language (Pelánek, 2007) here a model is composed of two main aspects; variables and processes.

5.1.1 Concurrent Systems as Variables and Actions

To use the benchmark protocols provided by the BEEM Library[1] DVE was chosen as an input model representation. The underlying theoretical model of the DVE language is that of communicating finite state machines and consists of several parts, structured hierarchically and identified as global variables and a number of processes on the top level.

The DVE language composes a model of a set of *processes* and a set of *variables*. Where a process is described by a variable representing the *state* of the process and a set of *actions*. A tuple of a *guard* and a set of *effects* denote an action also called *transition*. In Model Checking the term guard is used to depict preconditions that must be true for an action to be active, while the term effect is used for postconditions. Reasoning on the transitions in Model Checking the state space can be reduced, e. g., using the partial order reduction (Godefroid, 1996) or data-flow analysis (Steffen, 1991).

Information sharing between processes and information storage is accomplished using variables.

Definition 26 (Variable) *A variable represents a container for information from a finite domain accessible either by all processes (*global *variable) or by one specific process (*local *variable). If a variable is defined as* constant *its information does not change between states.*

Each variable can be identified by its name denoted in the model and needs to be stored in the state, since it is required to describe the condition the system is in. Usually variables are stored as a bit vector of adjustable size in the state and a table links its name to an index position. This work restricts for finite system requesting for variables from a finite domain, infinite systems are described in detail by Burkart and Steffen (1997).

The umber of processes is not limited but only one transition in each process is evaluated in the DiVinE model checker. While in sequential models only one process

[1] see: http://anna.fi.muni.cz/models/

is active in one state parallel models activate all processes evaluating a number of transitions at once.

The description language used to describe models in the model checker DIVINE adds an explicit local state to each process called *process state*. The system state is then composed of a number of process states accordingly to Definition 3 of a local state in the introduction.

Definition 27 (Process State) *Each process in a specification resides in a* process state *defined prior to the search. One process state has to be marked as an initial process state which is the state the process is in when the search starts.*

The process state is realized as an additional private variable assigned to the process. Each transition in Model Checking is extended by an additional guard checking whether the process is in the process state the transition expects. The modification of the process state is realized in the postconditions of the transition. The remaining parts of the transition are defined exactly like in the introduction by using preconditions, denoted as guards and postconditions denoted as effects. In contrast to only allowing one guard the number of effects is not bounded.

Two different approaches exist to traverse the graph generated from the representation, a *two-passes* Model Checking, where the state space is first generated, then checked, and *on-the-fly* Model Checking (Courcoubetis *et al.*, 1992; Fernandez *et al.*, 1992). Here the checking process is performed in parallel to the generation.

5.1.2 Explicit State Model Checking

In *Explicit State Model Checking* a system of transitions is checked against a given property. This work is based on the description of the model in *Linear Temporal Logic* (LTL) which uses a temporal logic for specifying the behavior of a software system (Pnueli, 1977). Automata-based LTL Model Checking (Vardi and Wolper, 1986) amounts to detecting accepting cycles in a global state space graph after converting the transition system and the property into Büchi automata. However, Wolper (1983) showed that a conversion is not always possible due to the increased expressiveness of Büchi automata. Denoting the automata of the model with $B(M)$ and the property with $B(\phi)$ the system satisfies the property if

$$\mathit{Lang}(B(M)) \subseteq \mathit{Lang}(B(\phi))$$

The system satisfies the property ϕ if all paths in the automaton M satisfy ϕ. Since such a definition would always require to check the whole state space the previous formula is converted to

$$\mathit{Lang}(B(M)) \cap \overline{\mathit{Lang}(B(\phi))} = \emptyset$$

equivalent to

$$\mathit{Lang}(B(M)) \cap \mathit{Lang}(B(\neg\phi)) = \emptyset$$

and

$$Lang(B(M) \cap B(\neg\phi)) = \emptyset$$

Now, every path of the combined automata is checked for not satisfying ϕ or, equivalent to this for satisfying $\neg\phi$. Such a checking process is sufficient for safety properties and can be realized by a simple BFS generation of the state space.Barnat *et al.* (2005) propose to invoke a second search from each BFS *backward edge* (connecting states to previous BFS-Layers), to detect cycles while in (Brim *et al.*, 2004) predecessor acceptance is chosen.

States which satisfy the properties are denoted as *accepting* states and a Büchi automata is accepting when a path exists from the initial state $\hat{s}$ to the accepting state s_a and s_a is part of a cycle. This leads to the definition of a *counterexample* in the form given in Figure 5.1 where s_a is visualized double circled. The state connecting the cycle to the path from initial is called *seed* state.

There are two possible criteria for *minimal counterexamples*. The external memory LTL Model Checking algorithm of Edelkamp and Jabbar (2006b) produces a minimal-length lasso-shaped counterexample $\tau_1\tau_2$ among the ones that include the accepting state at the seed state of the cycle (Figure 5.1 left).

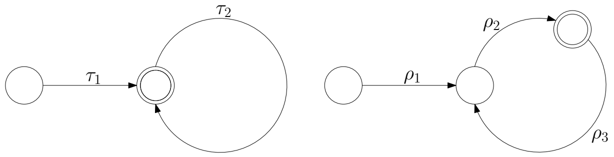


Figure 5.1: Different optimality criteria for a lasso-shaped LTL counterexample.

For this work, a stronger optimality criterion is assumed, in which the counterexample $\rho_1\rho_2\rho_3$ (Figure 5.1 right) is minimal among all lasso-shaped counterexamples (not necessarily having an accepting state at the seed of the cycle). It is obvious that the length is at most as large as the above one.

5.1.3 Explicit State Model Checking Example

Figure 5.2 models the graph in Figure 1.1 on Page 7 in the DVE language (Pelánek, 2007) specification. An integer global variable `comlpeteness` is defined carrying the progress of the thesis and accessible by all processes defined in the following code. The first process called `student` contains two private Boolean variables called `sleepy` and `hungry` instantiated to `true`. The process can be in four process states, specified at the beginning with the word `state` and being `sleeping`, `eating`, `thinking` and `writing` where the initial process state is `sleeping`. Transitions are introduced by the word `trans` and grouped in curly brackets. Each transition starts with the process state it requires followed by an arrow `->` and the process state the process will be in after evaluation. Guards and effects are prefixed by the appropriate key and separated in case of the effects by a semicolon. Analogous to the first process the `friend` process is modeled containing two process states and two transitions. The third process is a special one defining the LTL property with an accepting state `q2` and

```
int completeness = 0;

process student {
  bool sleepy = true;
  bool hungry = true;

  state sleeping, eating, thinking, writing;
  init sleeping;
  trans{
    sleeping -> eating {effect sleepy = false},
    eating -> thinking {effect hungry =false},
    thinking -> writing {},
    writing -> thinking {guard !sleepy and !hungry;
                    effect hungry;completeness+=10;},
    writing -> eating {guard hungry;
                    effect sleepy;completeness+=10;},
    writing -> sleeping {guard sleepy;
                    effect hungry;completeness+=10;}
   };
}

process friend{
   state enjoingTime, reading;
   init enjoingTime;
   trans{
     enjoingTime -> reading {guard completeness > 0;},
     reading -> enjoingTime {effect completeness --;}}
}

process LTL_property {
   state q1, q2;
   init q1;
   accept q2;
   trans{
     q1 -> q1 {},
     q1 -> q2 {guard completeness >= 100;},
     q2 -> q2 {};
   }
}
```

Figure 5.2: Graph in Figure 1.1 on Page 7 described as a DVE model.

only one guard in the transitions being enabled when `completeness` has reached 100%.

5.2 Related Work

A detailed overview in the area of Explicit State Model Checking can not be given in the scope of this thesis so the next sections will sketch the most important work done in the fields of External Explicit State Model Checking and Parallel Explicit State Model Checking.

5.2.1 External Explicit State Model Checking Algorithms

Edelkamp and Jabbar (2006b) present the first I/O-efficient solution for the LTL Model Checking problem which builds on the reduction of liveness-to-safety property conversion from Schuppan and Biere (2004), originally designed for symbolic Model Checking (McMillan, 1993). The algorithm operates on-the-fly and applies heuristics (Pearl, 1985) for accelerated LTL property checking. Since the exploration strategy is A* (Hart *et al.*, 1968), the produced counterexamples are optimal.

A further I/O-efficient algorithm for accepting cycle detection (Barnat *et al.*, 2007) is one-way-catch-them-young (OWCTY). It generates the whole state space and then iteratively prunes parts that do not lead to any accepting cycle. Later on, an external on-the-fly LTL Model Checking algorithm based on the maximal-accepting-predecessors algorithm (MAP) (Barnat *et al.*, 2008b) and nested Depth-First searches (Fernandez *et al.*, 1992) has been developed.

Edelkamp *et al.* (2004b) coin the phrase *Directed Model Checking* to denote a guided traversal of the state space supported by heuristics by implementing the guided explicit-state model checker HSF-SPIN, an extension to SPIN (Holzmann, 2004). While *Approver*, proposed by Jan (1978) was already a too that used a directed search for the verification of communication protocols, *SpotLight* (Yang and Dill, 1998) applied the basic AI algorithm A* (Pearl, 1985) for the verification of models. Based on this approaches were invented to utilize heuristics in Model Checking, e. g., by Edelkamp *et al.* (2004a) and by Jabbar and Edelkamp (2005). Followed by external memory and even a parallelized algorithm to decrease the searching time (Edelkamp and Jabbar, 2006b; Jabbar and Edelkamp, 2006; Edelkamp and Schroedl, 2011). Both algorithms do utilize external memory but not necessarily the advantage of solid state media which would provide an additional speed up.

5.2.2 Parallel Explicit State Model Checking

The first appearance of distributed Model Checking goes back to Aggarwal *et al.* (1987) who investigate aspects of distributing reachability over a local area network of workstations, in order to reduce the time needed to complete the calculation. However, the algorithm was never implemented so that the first practical usage of distributed Model Checking was by Stern and Dill (1997). They describe a parallel version of the explicit state enumeration verifier Murϕ for distributed memory multiprocessors and networks of workstations using the message passing paradigm. An approach like presented in (Edelkamp *et al.*, 2008a) is not suitable due to necessary communication between threads.

Currently a number of Model Checking tools exist being extended to utilize parallel hardware in a shared or distributed memory architecture. DIVINE, a tool utilized also in the scope of this work, is a parallel shared memory LTL model checker that is based on a distributed memory algorithm. A number of implementation techniques presented in (Barnat *et al.*, 2010b) is devised to improve the scalability of the tool.

In Holzmann *et al.* (2008) one of the most prominent Model Checking tools, namely SPIN (Holzmann, 2004) is parallelized by using the *swarm intelligence* technique described by Hofstadter (1979).

The LTSmin tool set (Laarman *et al.*, 2011a) provides multiple generation and on-the-fly analysis algorithms for large state spaces in symbolic and distributed Model Checking algorithms. Recently a multi-core back end for checking safety properties was added (Laarman *et al.*, 2011b), improving efficiency and memory usage .

Barnat *et al.* (2009) present a tool that performs CUDA accelerated LTL Model Checking. They adjust the MAP algorithm to the GPU to detect the presence of accepting cycles. As in bounded Model Checking (Biere *et al.*, 1999), the state space may be generated in layers on the CPU, before being transformed into a matrix representation to be processed on the GPU. The speed-ups are visible, but the approach is limited by the memory available on the GPU and able to checking properties in moderately-sized models only. The approach is modified to utilize multiple devices (Barnat *et al.*, 2010a) extending the possible model size by a factor of two due to using two cards.

5.3 Summary

As a starting point this chapter introduces Explicit State Model Checking and the modeling of concurrent systems. The input language DVE discussed in detail and an example of a model was given. The chapter closes with a presentation of related work on Explicit State Model Checking.

Chapter 6

SSD-Based Minimal Counterexamples Search

Many scenarios with the claim for a minimal counterexample exist in Model Checking. Be it in the case where a non minimal model checker finds a counterexample to long to reconstruct for the developer or when even the minimal one is known to have a decent length. In such situations investing more time or other resources in the checking process may pay off when analysing the returned example. This chapter will propose an algorithm which traverses a the state space partially or even completely three times but ensures to deliver a minimal counter example. The preliminaries of this chapter first appeared in (Edelkamp and Sulewski, 2008a) extended a refined in (Edelkamp and Sulewski, 2008b). Later on this approach is presented in (Edelkamp *et al.*, 2011) together with different other techniques for using the SSD in hashing and Model Checking.

Structure of this chapter: Starting with the definition of semi-external algorithms this chapter introduces the baseline algorithm from Gastin and Moro (2007) extended to support solid state media. In the next section the modification to use perfect hashing to externalize the majority of information is described.

6.1 Semi-External LTL Model Checking

With limited information per state (e. g., one flag for monitoring if a state has been visited), semi-external graph algorithms (Abello *et al.*, 1998) store a constant number of bits per state.

Definition 28 (c-bit semi-external graph algorithm) *A graph algorithm $\mathcal{A}$ is called* c-bit semi-external *for $c \in \mathbb{R}^+$, if there is a constant $c_0 > 0$ such that for each implicit graph $G = (V, E)$ the internal memory requirements of $\mathcal{A}$ are at most $c_0 \cdot v_{max} + c \cdot |V|$ bits. Including the state vector size v_{max} in the complexity is necessary, since this value varies for different graphs.*

The worst case I/O complexity of the presented algorithm is roughly $|Accept|$ times the one of semi-external BFS with internal duplicate detection. Its shows a considerable improvement to (Edelkamp and Jabbar, 2006b). More importantly, the worst-case space consumption is linear in the model size and matches the ones of OWCTY and MAP.

Based on the definition of perfect hash functions given in the introduction, semi-external Depth-First search with the use of minimum perfect hashing has been proposed (Edelkamp *et al.*, 2008b) . First, an external memory BFS (Munagala and Ranade, 1999) generates all states on disk (the external step). Then, a minimal perfect hash function (residing in RAM) is constructed on the basis of all these states. Finally, the actual verification is performed using this perfect hash function to address a 1 bit table (see Figure 6.1) for the $Closed$ state set.

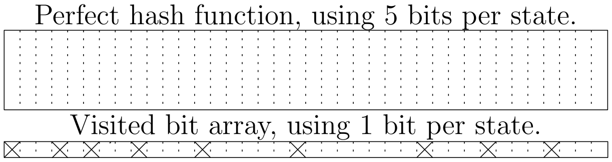

Perfect hash function, using 5 bits per state.
Visited bit array, using 1 bit per state.

Figure 6.1: State space compression utilizing a perfect hash function which compresses a state to 5 bits (the position of a visited bit is determined by the state's compressed representation in the function).

In a related publication the double Depth-First search algorithm originally proposed by Courcoubetis *et al.* (1992) has been ported from internal to external search for semi-external LTL Model Checking. The algorithm performs a first DFS to find all accepting states. The second DFS explores the state space seeded by these states. Besides the amount of space for storing the perfect hash function, the algorithm requires one additional bit per state.

Depth-First search based algorithms (usually) produce non-minimal counterexamples. Many other algorithms, like OWCTY and MAP, do also not guarantee minimality of the counterexamples produced.

The following implementation, calling a semi-external BFS $O(|Accept|)$ times, adapts the algorithm of Gastin and Moro (2007), an internal-memory algorithm that finds optimal counterexamples space-efficiently.[1] The algorithm progresses only along forward edges.

Algorithm 6.1 provides the pseudo-code for the search for a minimal counterexample with a combination of solid state and hard disks. The construction of the priority queue[2] is shown in Algorithm 6.2, while the synchronized traversal is shown in Algorithm 6.3.

Gastin and Moro (2007) propose three phases, corresponding to the three concatenated sub-paths of the counterexample $\rho_1\rho_2\rho_3$, show in Figure.5.1. Path ρ_1 to the cycle seed (phase 1), path ρ_2 from the seed to the accepting state (phase 3) and path ρ_3 back

[1] In (Gastin and Moro, 2007), the algorithm was not implemented. Hence, this presentation eliminates some minor bugs.

[2] The notation aligns with (Gastin and Moro, 2007), proofs of correctness and optimality are inherited.

Algorithm 6.1: Minimal-Counterexample search

```
Input: $\hat{s} \in \mathcal{S}$ initial state
Output: Minimal Counter Example if exists
$(\mathcal{S}, \epsilon_{\hat{s}}) \leftarrow$ External-BFS$(\hat{s})$ ;   {start External BFS from $\hat{s}$ }
    {generate $\mathcal{S}$ and store the maximal depth in $\epsilon_{\hat{s}}$}
$h \leftarrow$ Construct-PHF$(\mathcal{S})$ ;   {construct the perfect hash function to $\mathcal{S}$ }
$Closed \leftarrow (0..0)$ ;   {internal bit-array of length $|\mathcal{S}|$}
$Open \leftarrow \emptyset$ ;   {file for state vectors on external memory}
$(depth, Accept) \leftarrow$ BFS-distance$(\hat{s}, Open)$ ;
    {find all accepting states and store their distance to $\hat{s}$ }
$opt \leftarrow \infty$ ;   {initial optimal lasso length}
for $s_a \in$ Accept $\wedge$ depth$(s_a) <$ opt do
{for each accepting state whose distance to $\hat{s}$ is smaller then opt}
    $Open \leftarrow \emptyset; Closed \leftarrow (0..0)$ ;   {clear $Open$ and $Closed$ list}
    $PQ \leftarrow$ BFS-PQ$(s_a, Open, Closed)$ ;
    {determine minimal $\rho_1 + \rho_3$ for all states reachable from $s_a$}
    $Open \leftarrow \emptyset; Closed \leftarrow (0..0)$ ;   {clear $Open$ and $Closed$ list}
    $(t, n) \leftarrow$ Prio-min$(s_a, PQ, Open, Closed)$ ;
    {determine minimal $\rho_2$ for all states with a path to $s_a$}
    if $(n <$ opt$)$ then
        $s_1 \leftarrow t$; $s_2 \leftarrow s_a$; $opt \leftarrow n$ ;

    {reconstruction of the counterexamples knowing $\rho_1, \rho_2$ and $\rho_3$}
$Open \leftarrow \emptyset; Closed \leftarrow (0..0)$ ;   {clear $Open$ and $Closed$ list}
$\rho_1 \leftarrow$ Bounded-DFS$(\hat{s}, s_1, Open, Closed, depth(s_1))$;
    {reconstruct $\rho_1$ using a DFS search bounded to $depth(s_1)$}
$Open \leftarrow \emptyset; Closed \leftarrow (0..0)$ ;   {clear $Open$ and $Closed$ list}
$dist(s_1, s_2) \leftarrow$ BFS$(s_1, s_2, Open, Closed)$;
;   {determine length of $\rho_2$ using a BFS search}
$Open \leftarrow \emptyset; Closed \leftarrow (0..0)$ ;   {clear $Open$ and $Closed$ list}
$\rho_2 \leftarrow$ Bounded-DFS$(s_1, s_2, Open, Closed, dist(s_1, s_2))$;
    {reconstruct $\rho_2$ using a DFS search bounded to $dist(s_1, s_2)$}
$Open \leftarrow \emptyset; Closed \leftarrow (0..0)$ ;   {clear $Open$ and $Closed$ list}
$\rho_3 \leftarrow$ Bounded-DFS$(s_2, s_1, Open, Closed, opt - depth(s_1) - dist(s_1, s_2))$;
{reconstruct $\rho_3$ using a DFS search bounded to $opt - depth(s_1) - dist(s_1, s_2)$ }
```

Algorithm 6.2: *BFS-PQ* File-based 1-level-bucket priority queue

```
Input: $s_a$ Accepting state, $Open$ file for state vectors
Output: PQ Dynamic array of state vector files (external)
if (depth($s_a$) $<$ opt) then                         {if depth of $s_a$ less then current opt}
    PQ[depth($s_a$) $+ 1$] $\leftarrow s_a$ ;          {append $s_a$ to array at position depth($s_a$)}
    $Open \leftarrow s_a$ ;                            {append $s_a$ at the end of $Open$ }
    $Closed[h(s_a)] \leftarrow$ true; ;                {mark $s_a$ as visited}
$n \leftarrow 0$ ;                                     {current distance to $s_a$}
loop $\leftarrow$ false ;                              {necessary to store whether $s_a$ was reached again}
$l \leftarrow 1$ ;                                     {number of states in current BFS-Layer started with $s_a$}
$l' \leftarrow 0$ ;                                    {counter for states in next BFS-Layer started with $s_a$}
$q \leftarrow 0$ ;                                     {read pointer for $Open$ }
while ($q \neq |Open| \wedge n <$ opt) do              {elements in $Open$ and $n$ shorter then opt}
    $u \leftarrow Open; q \leftarrow q + 1; l \leftarrow l - 1$ ;   {read state, increase $q$, decrease $l$}
    expand $u \rightarrow s_1 \ldots s_\nu$ ;          {generate successors}
    for $s_i$ ($\forall i : 1 \leq i \leq \nu$) do     {for each successor of $u$}
        if ($Closed[h(s_i)] =$ false) then             {if $s$ is an unseen state}
            $Closed[h(s_i)] \leftarrow$ true ;         {mark $s_i$ as visited and store it in $Open$ }
            $Open \leftarrow s_i; l' \leftarrow l' + 1$ ;
            {append $s_i$ to $Open$, increase next layer counter $l'$}
            if (depth($s_i$) $+ n + 1 <$ opt) then
            {if length of a counterexample crossing $s_i$ is smaller then opt}
                PQ[depth($s_i$) $+ n + 1$] $\leftarrow s_i$ ;   {add it to appropriate PQ list}
        loop $\leftarrow$ loop $\vee$ ($s_i = s_a$) ;  {when cycle found remember this}
        if (loop $\wedge$ depth($s_i$) $+ n + 1 <$ opt) then
            opt $\leftarrow$ depth($s_i$) $+ n + 1$ ;  {store length of counterexample in opt}
    if ($l = 0$) then                                  {if all states in this BFS-Layer done increase $n$}
        $l \leftarrow l'; l' \leftarrow 0; n \leftarrow n + 1$ ;
if loop then return PQ ;
else return $\emptyset$ ;
```

Algorithm 6.3: Prio-min: Synchronized traversal driven by an external memory 1-level-bucket priority queue.

```
Input: $s_a$ Accepting state , $Open$ file for state vectors, PQ dynamic array of
       state vector files
Output: Pair $(u, t)$ of state $u$ and lasso length $t$
$n \leftarrow \min\{i \mid PQ[i] \neq \emptyset\}$ ;                {start with first non empty PQ}
$p \leftarrow \sum_i |PQ[i]|$ ;                                    {counter for all states in all PQ files}
$q \leftarrow 0$ ;                                                  {read pointer for $Open$ }
$l \leftarrow 0; l' \leftarrow 0$ ;                                 {counters for current and next BFS-Layer}
while $((p \neq 0 \vee q \neq |Open|) \wedge (n + 1 \neq \text{opt}))$ do
{until all states processed or $Open$ empty}
    for $(s \in \text{PQ}[i] \wedge Closed[h(s)] = \texttt{false})$ do
    {get all states from current PQ}
        $Open \leftarrow (s, s); Closed[h(s)] \leftarrow \texttt{true}$ ;    {insert a pair into $Open$ }
        $p \leftarrow p - 1; l \leftarrow l + 1$ ; {decrease state and increase BFS-Layer counter}
    while $(l \neq 0)$ do                                           {while states to process}
        $(s, s') \leftarrow Open; q \leftarrow q + 2$ ;             {get two states from $Open$ }
        expand $s' \rightarrow v_1 \ldots v_\nu$ ;                  {generate successors}
        for $v'_i$ $(\forall i : 1 \leq i \leq \nu)$ do             {for each successor of the second state}
            if $(v'_i = s_a)$ then                                  {if a path to the accepting state found}
                return $(s, n + 1)$ ;                               {return $s$ and length of counterexample}
            if $(Closed[h(v'_i)] = \texttt{false})$ then            {state is unvisited}
                $Closed[h(v'_i)] = \texttt{true}$ ;                 {mark it as visited}
                $Open \leftarrow (s, v'_i)$ ;                       {add a new pair to $Open$ }
                $l' \leftarrow l' + 1$;
        $l \leftarrow l - 1$ ;                                      {decrease BFS level counter}
    $l \leftarrow l'; l' \leftarrow 0; n \leftarrow n + 1$ ;        {change $l$ to $l'$ and reset $l'$}
return $(\perp, \infty)$ ;
```

from the accepting state to the seed (phase 2). Phase 1 of the algorithm executes a plain BFS, that comes for free while constructing the perfect hash function, even though the implementation performs another semi-external BFS for it. Phase 2 and 3 start a BFS from each accepting state, incrementally improving a bound *opt* for the length of the minimal counterexample. Phase 3 invokes a BFS driven by an ordering obtained by adding the BFS distances from phases 1 and 2. States in this phase are ordered with respect to $|\rho_1| + |\rho_3|$ and stored in a 1-level bucket priority queue, originally invented by Dial (1969).[3] If duplicate elimination is internal, states can be processed in sequence. Hence, all three phases do allow streaming and can be implemented I/O-efficiently.

Files are organized in form of queues, but they do not support the *Dequeue* operation, for which deleting and moving the content of the file would be needed. Therefore, instead of *Enqueue* and *Dequeue* the algorithms are rewritten based on the operations

[3] In (Gastin and Moro, 2007), a heap was used, which is less efficient.

Append and *Next*. As a consequence that files do not run empty, and in order to keep the data structures on the external device as simple as possible, the implementation had to be adopted. The counters l and l' maintain the actual sizes of the currently active and the next BFS-Layer (at the end of a layer, l' is set to 0 counting the unique successors of the next layer). Value q denotes the current head position in the queue file and is incremented after reading an element. When the queue file is scanned completely, elements are removed and q is set to 0.

With the constant access time, perfect hashing speeds up all graph traversals to mere scanning. It provides duplicate detection and fast access to the BFS depth values (wrt. the initial state) that have been associated with each state. Finally, solution extraction (slightly different to Gastin and Moro (2007)) reconstructs the three minimal counterexample sub-paths ρ_1, ρ_2, and ρ_3 between two given states using bounded DFS. In difference to Gastin and Moro (2007) the *depth* value is not overwritten. DFS depth is determined by the stack size, such that, once the threshold is known, no additional pruning information is needed.

Counterexample reconstruction based on bounded DFS can also be implemented semi-externally. Let *opt* be the length of the optimal counterexample and *dist* the length of the shortest path between two states. It is not difficult to see that for start state s, seed state s_1, and accepting state s_2 we have $|\rho_1| = \mathit{dist}(s, s_1)$, $|\rho_2| = \mathit{dist}(s_1, s_2)$ (to be computed with BFS), and $|\rho_3| = \mathit{opt} - \mathit{dist}(s, s_1) - \mathit{dist}(s_1, s_2)$. The reconstruction is slightly different to Gastin and Moro (2007) as the knowledge on $|\rho_1|$ and *opt* to avoid BFS calls is used.

The implementation includes performance improvements mentioned by Gastin and Moro (2007), while constructing the priority queue. For example, accepting states without loops or over-sized loops are neglected. If the first loop established (including the depth of the seed) is already too big, the construction terminates. Moreover simple loops on the accepting state are filtered.

The upper size for the priority queue is bounded by the maximum depth ϵ_s of the BFS starting at s, plus the diameter of the search space $\mathit{diam} = \max_{s_1,s_2} \mathit{dist}(s_1, s_2)$. As the latter is not known in advance, dynamic vectors for storing the priority queue are needed.

6.1.1 Extending to Efficiently Support SSDs

The key idea to improve the RAM-efficiency of semi-external memory algorithms is rather simple. Instead of the hash function being maintained completely in RAM, it is stored (partially or completely) on the solid state disk. $Closed$ remains in RAM and consumes one bit per state.

Note that *static* perfect hashing, as approached in this chapter (in contrast to *dynamic* (perfect) hashing[4] (Barnat *et al.*, 2008a)) is flash-efficient. Most perfect hashing algorithms can be made dynamic (Dietzfelbinger *et al.*, 1994), but on SSDs the additional limitation of slow random writes exists, so that rehashing has to be sequential. In other words, foreground and background hash functions have to be compatible. The

[4] where, each time RAM becomes sparse, the foreground function, which stores the states in the RAM, has to be moved and merged with the background hash function, using external storage

design of flash-efficient dynamic hashing algorithms is a research challenge on its own with a large impact for on-the-fly Model Checking.

The setting distinguishes three phases: state space generation, perfect hash function construction and search. The external memory construction process used by Botelho and Ziviani (2007) is streamed and includes sorting the input data according to some first-level hash function. Therefore, it can be executed efficiently on the hard disk. In preceding experiments with a key set provided as a file, the perfect hash function was constructed also in form of a file, reading the keys from disk and writing the generated perfect hash function to it. Or from hard disk to solid-state disk, or from flash media card to solid state disk. The compression ratio is impressive, e. g., for sets of 10-letter strings, an 18-fold reduction is obtained.

Figure 6.4 shows the extended algorithm with integrated flash memory for storing and accessing the perfect hash function. It requires one bit per state for early duplicate detection in RAM. For calling *Semi-External-LTL-Model-Check*, different options are available. For the example of single or double Depth-First search (possibly combined with iterative-deepening), one bit per state in $Closed$ is sufficient.

Algorithm 6.4: SSD-LTL-Model-Check: Flash-efficient semi-external Model Checking

```
$\textit{State Space} \leftarrow \textit{External-BFS}(s)$ ;
$h$ : Perfect hash function, $c_{PHF} \times |\textit{State Space}|$ on SSD ;
$h \leftarrow \textit{Construct-PHF}(\textit{State Space})$ ;
$Closed$ : internal bit-array $[1..|\textit{State Space}|]$ ;
$\textit{Semi-External-LTL-Model-Check}(s, h, Closed)$ ;
```

Exploiting flash memory, the semi-external minimal counterexample algorithm described above can be made 1-bit semi-external, if the BFS depth is attached to the state in the perfect hash function on the solid state disk. Therefore, the number of bits required at each state on solid state disk is enlarged by the logarithm of the index of the maximum BFS-Layer. Assuming that this value is smaller than 256, which was the case in our experiments, one byte per state is sufficient.

6.2 Externalizing the Perfect Hash Function

To understand the externalization of minimum perfect hashing, it is necessary to motivate its construction process and its usage. Perfect hashing as defined in Section 1.4.1 is an one-to-one mapping of some state set V to the index range $\{0, \dots, |V| - 1\}$. For the construction of the hash function, the set V has to be known.

For a global state lookup, perfect hashing requires 1 seek, then reading a sequence of bits, depending on the implementation. If the number of bits is smaller than the block size, besides multiple calls to the read operation no additional overhead is required.

External perfect hashing (Botelho and Ziviani, 2007) builds on a partition with buckets of at most $n = 256$ elements each, using a first-level hash function that guarantees 128 bucket elements on the average, and no more than 256. For each of the buckets, two individual hash tables exist on which a bipartite graph is built. The two hash functions of each bucket can be stored compactly in $m = 2c_{PHF}n$ bits, with $c_{PHF} \approx 1.05$. If the number of elements in the addressed bucket by a first-level hash function, is 256, then $m \approx 530$ bits have to be stored to evaluate the perfect hash function correctly.

Two externalizations were implemented in the scope of this work. In the first one, only the information on the m bits is flushed, which leaves about 184 remaining bits in the RAM. In the second one, all information is flushed except the file pointer to the bucket, which reduces the number of bits per bucket to 64. In all implementations with access to the perfect hash function on flash memory, some information of the bucket remains in RAM, but beats the lower bound of 1.44 bits per state (Dietzfelbinger *et al.*, 1994). It is rather simple to extend the implementations to externalize the remaining bits to the disk by using a sparse representation of the buckets. A drawback of writing the uncompressed representation of the buckets to disk is that the file size increases by about a factor of 2 (from 128 on the average to 256). Avoiding this a 1-bit semi-external algorithm is constructed. Such 1-bit semi-external algorithm allows using almost all available RAM for $Closed$. The number of hard disk I/Os does not change. Semi-external LTL Model Checking is dominated by BFS state space generation.

Storing the hash function after generating the state space on hard disk, requires $\textit{write}(|V|)$ flash memory I/Os. During (double) depth-first search, for each state space edge, a query to the hash function is proposed, such that $O(|E| \cdot \textit{read}(1))$ flash memory I/Os are needed. As the hash function for the on-the-fly variant is computed for each BFS level, the flash memory complexity can increase.

For minimum counterexample generation, the following situation arises. If allocating $1 + c_{PHF} + \lceil \log(\epsilon_s + 1) \rceil$ bits per node exceeds RAM, flash memory helps. Outsourcing the perfect hash function together with the BFS-level takes $O(\textit{write}(|V|))$ flash memory I/Os, while total I/O complexity for the lookups for duplicate detection is bounded by $|\text{Accept}| \cdot |E| \cdot \textit{read}(1)$ I/Os. Storing the array *depth* on the solid state disk by enlarging the disk representation of the perfect hash function does not yield additional I/O, as the access to one compressed state (with depth value included), is still below the block size. Here, the access in the pseudo-code would change from $\textit{depth}(v)$ into $\textit{depth}[h(v)]$.

6.3 Summary

This chapter proposed a semi-external algorithm utilizing perfect hashing to store a portion of the information, necessary to traverse a graph on solid state media. Based on the internal minimal counterexamples algorithm from Gastin and Moro (2007) an additional BFS was inserted at the front to generate a perfect hash function. The algorithm performs 3 stages to generate the minimal counterexample, the generation of the perfect hash function using an external BFS, the search for the counterexample using internal memory and an SSD, and the reconstruction of the counterexample. The perfect hash function is generated and stored efficiently on SSDs and profits from the increased random access time of this media while searching. Having externalized it each BFS traversal can be done using the internal memory filled nearly completely with a 1-bit $Closed$ list supported by the SSD.

Chapter 7

GPU-Based Model Checking

In the following, the strategies presented in Part I are used for breadth-first explicit-state Model Checking on the GPU. This chapter will show how to test enabledness for a set of states in parallel, and – given all sets of applicable transitions – how to generate the successor state sets accordingly. BFS for generating the entire search space is sufficient for verifying the safety properties. Even for Model Checking full LTL, presented in the previous chapter. Efficient state space generation via Breadth-First search is often a crucial step. The external memory evaluation was published in (Edelkamp and Sulewski, 2010b) and the internal approach in (Edelkamp and Sulewski, 2010a).

The state space generation is divided into three stages according to the framework propsed in Part I. In the first stage, a set of enabled transitions is generated by copying the states to the VRAM and replacing them by a bitvector using the successor pointing strategy described in detail in Section 4.2.2. In the second stage, sets of all possible successors are generated. For each enabled transition a pair, joining the transition ID and the explored state, is copied to the VRAM. Each state is replicated by the number of successors it generates in order to avoid memory to be allocated dynamically. After the generation all duplicates are removed.

Structure of the chapter: Having described the extraction of a GPU suitable state out of a DVE model the checking of a formula is characterized. The chapter continues with details on the precondition checking and enabling of the postconditions to generate successors and closes with the comparison of internal and external duplicate detection.

7.1 Parsing the DVE Language

Based on the grammar knowledge, the model description can be parsed and a syntax tree constructed. To store different variable assignments and indicate in which state a process currently is, a byte vector can be used. Figure 7.1 describes the state vector assigned to the example in Figure 5.2. Necessary space for each global variable is reserved, followed by the current state of a process, represented as an integer, and

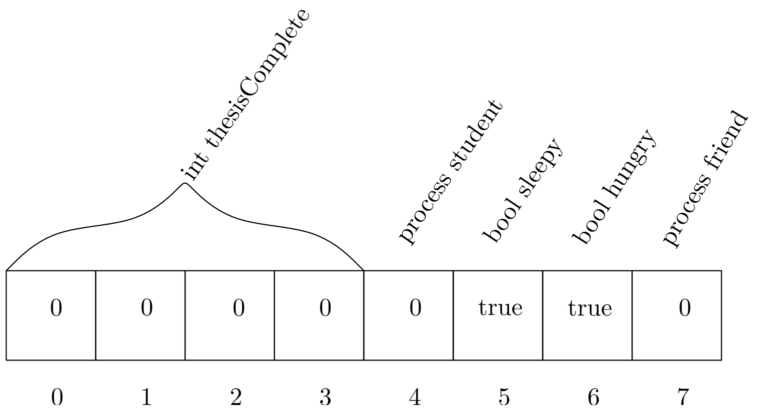


Figure 7.1: State vector representing a state with a global 32-bit integer denoted as *thesisComplete*. One process whose process state is stored at position 4 and whose private variables are *sleepy* and *hungry* followed by a second process called *friend*.

combined with space for all its local variables.[1]

Converting the protocol to the Reverse Polish Notation and copying it to the GPU is executed before the Model Checking process starts. Using this representation a check for enabledness of a transition in a process boils down to 3 steps:

1. Checking the state the process is in, by reading the corresponding byte in the state vector.

2. Identify transitions to check by reading the global prefix of the integer vector describing the model.

3. Evaluation of all guards dependent to the actual state and process on a stack.

To enable a transition given its ID, the representation of its effects starting at the position given in the second partition of the array has to be evaluated. The advantage of this approach is to copy all information needed for the model checking process into 1 block. Given that all guards and effects, respectively, are located in adjacent memory cells, a streamed access for evaluating a large number of guards is realized.

Figure 7.2 picks up the vector in Figure 4.3 and extends it to two transitions. As proposed in Section 4.5 of Part I the guards are rewritten into the Reverse Polish Notation. Moreover, additional static information about the structure of the postfix representation, needed to evaluate a guard is copied to separate memory blocks.

This information includes, e. g., the offset of the guards for each process and the starting position of guards depending on the state a process is in. For the application of a transition to a given state, similar to processing the guards, the effect expressions have been also rewritten in Reverse Polish Notation. Since this static representation resides in the GPU's VRAM for the entire checking process and since it is addressed by all

[1] This representation is equivalent to the one used in the DiVinE model checker.

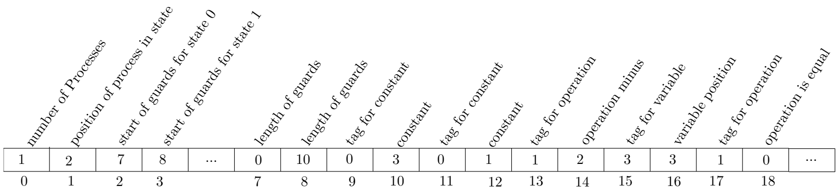


Figure 7.2: Fragments of the transition vector to evaluate two transitions. The first transition, starting at index 7 comes without a precondition and the seconds transition precondition is *my_place == 3-1* whose length is 10, denoted at position 8.

instances of the same kernel function, its access is fast. The cause is that broadcasting is an integral operation on most graphics cards.

7.1.1 Checking Enabledness on the GPU

Before the execution the transition vector is copied to the SRAM for faster access. All threads access in parallel the VRAM and read the state vector into their registers using coalescing. Then all threads access *transitions*[0], the vector visualized partly in Figure 7.2, to find the number of processes in the model. Next, all threads access *transitions*[1] to find the state the first process is in. At this point in time, the memory access diverges. Since processes have reached different states at different positions in the search space, different guards have to be evaluated. This does not harm, since the transition vector is accessible in the SRAM and all access is streamed. After collecting the necessary information, all threads call Algorithm 7.1 as the function in line 5 of Algorithm 4.4 (page 69). A stack consisting of pair entries (token, value) is used to evaluate the Boolean formulas. The maximal stack size is fixed for each protocol and can be extracted from the model.The checking process boils down to storing the values on the stack, and executing all operations on the two entries on top of the stack. The stack serves as a cache for all operations and if an assignment is found, the value on top of it is written to the state.

In the first stage the VRAM is filled with states from the $Open$ list. Then, according to the successor pointing strategy a GPU kernel computes a bit vector B of transitions, with bit B_t denoting, whether or not transition t applies. The entire array B, whose size is equal to the upper bound of all transitions, is initialized to `false`. A bit is set, if a transition is enabled. Each thread reads one single state at an unique position defined by its ID and computes the set of its enabled transitions. For improved VRAM efficiency the vector of transitions replaces the states they are applied to. Therefore, we utilize the fact that the number of transitions in a protocol is constant and the number of transitions does not exceed the size of the bit vector representation of a state. For the implementation, after having checked all transitions for enabledness, the bit vectors are copied back to RAM.

To evaluate a postfix representation of a guard, one scan through its representation suffices. The maximal length of a guard times the number of groups thus determines the parallel running time, as for all threads in a group, the check for enabledness is

Algorithm 7.1: *GPU-Kernel* Determine guard on a given state

```
Input: transitions expression vector, $s$ state in vector representation, tID
       transition ID
Output: true, if guard evaluation was successful; false, otherwise
pos $\leftarrow$ start_of_guard(tID) ;                    {set starting position of guard.}
while pos $<$ (start_of_guard(tID) $+$ length_of_guard(tID)) do
{while end not reached}
    if is_constant(transitions[pos]) then
        push transitions[pos+1] on top of stack;
                                          {constant? Just store it on the stack}
    if is_variable(transitions[pos]) then
        push state[transitions[pos+1]] on top of stack;
                    {variable? Read indexed value in $s$ and store it on the stack}
    if is_operator(transitions[pos]) then
        pop var1 and var2 from stack;
                                          {operator? Get two values from the top}
        result $\leftarrow$ var1 transitions[pos+1] var2 ;    {apply the indexed operator}
        push result on top of stack ;             {and store the result on the stack}
    pos $\leftarrow$ pos $+$ 2; ;                             {set pointer to the next element}
return result ;
```

executed concurrently.

Algorithm 7.2: *GPU-Kernel* Detect Duplicates via Sorting

```
Input: $H$ (unsorted)
Output: $H$ (partially sorted)
for each group $g$ do in parallel               {partially distributed computation}
    $i \leftarrow$ Select$(H, g)$ ;                              {transfer block to SRAM}
    $H'[i] \leftarrow$ ParallelSort$(H[i])$ ;                      {Sort using all threads}
return $H'$
```

7.2 Generating the Successors on the GPU

Having fixed the set of applicable transitions for each state, generating the successors on the GPU is relatively simple. First, each state is replicated to be explored by the number of enabled transitions on the CPU. Moreover, we attach the ID of the transition that is enabled together with each state. Then, the array of states is moved to the GPU and the successors are generated in parallel.

For the application of a transition to a given state, similar to processing the guards, the effect expressions have been rewritten in Reverse Polish Notation, and are evaluated according to the successor pointing strategy. Since this static representation resides in the GPU's VRAM for the entire checking process and since it is addressed by all instances of the same kernel function, its access is fast. The cause is that broadcasting is an integral operation on most graphics cards.

Each state to be explored is overwritten with the result of applying the attached transition, which often results in small changes to the state vector. Finally, all states are copied back to RAM. The run-time is determined by the maximal length of an effect times the number of groups, as for all threads in a group we generate the successors in parallel.

7.3 Duplicate Detection

Two strategies were used to perform the check for expanded states. An immediate detection on the CPU, checking all states in the set, and a delayed duplicate detection utilizing the GPU.

7.3.1 Immediate Detection on (Multiple Cores of) the CPU

Due to the fact that successors may be generated several times in one buffer this is not a strict immediate duplicate detection but a semi-immediate one. Since the successors are generated in parallel, an efficient parallel method is necessary to detect duplicates by checking the current state against the list of explored nodes. Like in the SPIN model checker, double Bitstate hashing is chosen as the default option. Looking at the number of states explored, the error probability for tens of gigabytes of main memory is acceptably small. Different options have been proposed to increase coverage (Holzmann, 1998), including the choice of a new hash function, e. g., from a set of universal ones (the state hash functions borrowed from Rasmus Pagh (Pagh and Rodler, 2001) are universal). To increase space utility, cache-, hash-, and space-efficient Bloom filters have been proposed (Putze *et al.*, 2009) and compress a static dictionary to its information-theoretic optimum by using a Golomb code. Refinements like sequential hashing with different hash functions, or hash compaction are possible but not yet implemented. To parallelize Bitstate hashing on multiple CPU cores, the set of successors is partitioned and all partitions are scanned in parallel as described in Section 4.3.2. In Bitstate hashing, a bit set is never cleared due to the nature of the BFS algorithm. States are never omitted in the parallel version of Bitstate hashing, however, since *Open* is stored externally and newer states always appended when considered new, a state can be added by two threads in parallel and expanded twice. This is not a problem since its successors would be caught by the duplicate detection and beside this fact, this never happened during the evaluation.

To avoid the probability of loosing a state in Bitstate hashing also table based hashing, storing the complete states in the table was evaluated but it could not keep pace with the Bitstate hashing approach in terms of time efficiency. So, having evaluated the state space numbers and noticed no discrepancy to Bitstate hashing this duplicate detection scheme was discarded.

7.3.2 Delayed Duplicate Detection on the GPU

Immediate duplication, being incredibly fast in the parallel implementation fills the available amount of RAM in about 15 minutes supported by the additional GPU processing power. Supporting larger state spaces and utilize the parallel power of the GPU also for the duplicate detection was considered and implemented in the way proposed in Section 4.3.1. Here $Open$ and also $Closed$ are stored entirely on external media preferring a RAID of devices to increase the access speed. The GPU is used for sorting in a bucket sorting approach utilizing the CPU to distribute the states, or representations of them in buckets sorted on the graphics card.

Due to the success when checking state spaces with state numbers over $2 * 10^9$ a compression strategy is used reintroducing omitting of states on a lower percentage level then the Bitstate approach. Beside being able to generate larger state spaces the time needed for sorting is still a significant speed killer compared with the hashing approach even when sorting is done on the GPU.

7.4 Summary

In this chapter the framework proposed in Part I was successfully applied to explicit Model Checking. After partitioning the search into three stages the successor pointing strategy was chosen to generate the states according to the given model and property converted to the Reverse Polish Notation. Several duplicate detection techniques were implemented to test the efficiency of immediate and delayed duplicate detection.

Chapter 8

Experimental Evaluation

The proposed algorithms were implemented in DIVINE (DIstributed VerIficatioN Environment)[1], including only part of the library deployed with it, namely state generation and internal storage. For the implementation of external memory containers and for algorithms for efficient sorting and scanning STXXL (Standard Template Library for Extra Large Data Sets) (Dementiev *et al.*, 2005) is used.

Since the developers of DIVINE switched from the 32-bit architecture to 64-bit during this work and changed the majority of DIVINE a complete reimplementation would be necessary to follow, a new model checker called CUDA Driven Model Checker (CUDMOC) was implemented.

Models are taken from the BEEM library (Pelánek, 2007), for which minimal counterexample lengths are not known.The capacity of the Bitstate table amounts to 81,474,836,321 bit entries with this number being a prime.

Structure of the chapter In the following the implementation of both algorithms is evaluated on different protocols taken from the BEEM library. Having examined the search for minimal counterexamples internally, the perfect hash function is externalized though more RAM remains for $Closed$. In the second part of this chapter GPU supported Model Checking is evaluated and compared to different state of the art model checkers. The chapter closes with a summary of this part.

8.1 Results for Semi-External LTL Model Checking

To get an idea on the achievable performance when checking a state space for minimal counterexamples an investigation in the algorithm running internally was performed. The results are presented in this section.

[1] http://anna.fi.muni.cz/divine

8.1.1 Minimal Counterexamples

First, the efficiency of the minimum counterexample generation on the 32-bit system is evaluated. To save time, we generate the state space on SSD, if possible. We observed a speed-up of a factor about 2, compared to the hard disk.

The first protocol used in a case study is Lifts (7) with the property 4. The state space consists of 7,496,336 states and 20,080,824 transitions and is generated in 262 layers. Its generation time amounts to about 1,122s on the SSD. The perfect hash function is first split into 58 parts, and then finalized in 66s. The first BFS that initialized the depth array and flushes the set of accepting states required 187s. The number of accepting states is 2,947,840. The minimum counterexample algorithm implementation first finds a counterexample with seed depth 81 and lasso length 117 (found within 10s), which is then improved to seed depth 87 and cycle length 2. Proving this to be optimal yields a total run time of 4,035s, with the CPU operating at 86%. According to Edelkamp *et al.* (2008b), the non-optimal semi-external double DFS approach takes about 1,920s to generate a counterexample. Factor 2.1 as a trade-off for optimality is acceptable, as optimality is an assertion about all paths.

For the Szymanski (4) model with property 2, the state space consists of 2,256,741 states and 12,610,593 transitions and is generated in 110 layers. Generation took 511s on the SSD. The hash function is split into 17 parts. The first BFS took 96s and generated 1,128,317 accepting states. The counterexample lengths found are $31 = 1 + 30$ and $19 = 2 + 11$. The last one is optimal. The total run-time is 2,084s. According to Edelkamp *et al.* (2008b), semi-external double DFS takes about 600s to generate a counterexample and is thus faster by about factor 3.4 only.

8.1.2 Flash-Efficient Model Checking

Table 8.1: Flash performance on double Depth-First search on models with invalid temporal properties (times are given in *mm:ss*).

	h in RAM		h on External Device		
	Amount	Time[mm:ss]	Amount	Time[mm:ss]	
Protocol	RAM		RAM	SSD	HDD
Szymanski (2), P3	28.77KB	0:06	2KB	0:50	0:37
Szymanski (3), P3	0.99MB	2:58	68.92KB	39:02	26:11
Lifts (7), P4	4.55MB	4:27	0.22MB	68:56	48:17
Lifts (8), P2	20.24MB	19:44	0.99MB	377:22	o.o.t.

Last, but not least, a look at the externalization of the hash tables to the SSD was taken. As the *depth* array was not yet externalized, LTL Model Checking with the double DFS implementation of Edelkamp *et al.* (2008b) is applied. Table 8.1 shows the results. First, the space consumption of the data structures for the models is reported, then compared to the time-space trade-off in three different externalizations. The first one stores the perfect hash function in the RAM, and thus matches the implementation of (Edelkamp *et al.*, 2008b). The other approaches externalize the hash function via direct I/O on either hard or solid state disk. Note that all experiments have a static storage offset of 238.89 MB due to the inclusion of the DIVINE model checker and STXXL.

Table 8.2: Space consumption of the tested instances.

	Space Consumption		
Protocol	*State Space*	h	$Closed$
Szymanski (2), P3	1.5MB	38.48KB	7.78KB
Szymanski (3), P3	65MB	1.33MB	275KB
Lifts (7), P4	351MB	5.67MB	915KB
Lifts (8), P2	1,559MB	25.21MB	4,071KB

The value *State Space* in Table 8.2 indicates the complexity of each model as stored on the hard disk. The number of states was not used here to highlight the compression ratio between the size of the state space and the size of the perfect hash function h since h is not dependent on the size of a state while the *State Space* is.

The columns h and $Closed$ show that the size of h is proportional to the size of $Closed$. Since $Closed$ is $|$*State Space*$|$ bits long and h contains a representation of each state, this is what one might have expected. Note that the Szymanski protocol needs 4.95 bits per state in the compressed form, while the Lifts protocol takes 6.34 bits per state on the average, as the implementation of Botelho and Ziviani (2007) is not capable to create a perfect hash function for this protocol using 4.95 bits per state.

For memory comparison between Edelkamp *et al.* (2008b) and the extension to it, the main memory usage for h is reported. The experiments show that the RAM usage drops by a factor of 14 for Szymanski and even by a factor of 20 for the Lifts protocol. As mentioned above, it is possible to externalize h completely, using more space on the external device, without an increase in I/O: the RAM usage is due to the fact that file-pointers are stored to every bucket in the h file (e. g., h[Lifts (8)] is stored in $260{,}579$ buckets, 127.99 entries per bucket in average, and a file-pointer is 4 bytes long which results in $1{,}042{,}316$ B = 0.99 MB) and could be omitted by imposing a constant bucket length in the file.

Outsourcing h on the external medium needs more time for the search and that for small experiments, where h fits into the hard disk cache, the externalization on hard disk is faster. Lifts (8), P2 is one experiment, where h is larger than 16 MB and does not fit into the hard disk cache. The experiment was stopped after six hours. During this time, the CPU usage never exceeded 5%, while the average CPU usage was 48% on solid state disk.

The time deficiency corresponds to a 19.1-fold slowdown with respect to Edelkamp *et al.* (2008b)[2] using $1/20$ of the main memory. Moreover, using swap space on solid state disk is prohibited by the operating system, so that hard disk is mandatory as a swapping partition. Time-efficiency is the main argument why to use solid state disk instead of hard disk for storing the perfect hash function in semi-external LTL Model Checking.

[2] Edelkamp *et al.* (2008b) is infeasible for very large model sizes.

Table 8.3: Experimental results, cross-comparing different versions of CUDA-driven model checker. Running times given in seconds.

	CuDMoC			
Protocol	Times in seconds			States
	1 Core CPU	1 Core + GPU	8 Core + GPU	
Anderson (6)	235	25	20	18,206,914
Anderson (8)	1381	669	440	538,493,685
At (5)	404	36	29	31,999,395
At (6)	836	170	119	160,588,070
Bakery (7)	296	30	28	29,047,452
Bakery (8)	3603	250	182	253,111,016
Elevator (2)	334	30	23	11,428,766
Fisher (3)	41	10	9	2,896,705
Fisher (4)	22	7	7	1,272,254
Fisher (5)	1692	126	86	101,027,986
Fisher (6)	107	16	13	8,321,728
Fisher (7)	4965	555	360	386,281,613
Frogs (4)	153	20	17	17,443,219
Frogs (5)	2474	203	215	182,726,077
Lamport (8)	867	70	49	62,669,266
Mcs (5)	896	77	50	60,556,458
Mcs (6)	12	7	7	332,544
Philosophers (6)	422	36	27	14,348,901
Philosophers (7)	2103	196	125	71,933,609
Philosophers (8)	1613	105	70	43,046,407

8.2 Results for GPU-Based Model Checking

Since internal Model Checking is much faster then the external approach two evaluation strategies were tested. In the first one CuDMoC is evaluated with state of the art internal Model Checking tools and in the following part it is compared mostly to itself due to the lag of comparable parallel and external Model Checking tools.

8.2.1 Evaluation of Immediate Duplicate Detection

The first evaluation in Table 8.3 analyzes the performance of the GPU algorithm compared to the CPU. The `--deviceemu` directive of the `nvcc` compiler was used to simulate the experiments on the CPU[3]. The table shows that using the GPU for the successor generation results in a mean speed-up (sum of all 1 Core + CPU times / sum of all 1 core + GPU) of 22,456 / 2,638 = 8.51. Column *8 Core + GPU* displays additional savings obtained by utilizing all 8 CPU cores for duplicate detection, operating simul-

[3] Earlier experiences showed no significant speed difference between simulating CUDA code with this directive and converting it by hand to, e. g., POSIX threads.

Table 8.4: Experimental results, comparing CUDA-driven model checker with DiVinE. (o.o.m. denotes out of memory)

	CuDMoC			DiVinE		
Protocol	Times in seconds		States	Times in seconds		States
	1 Core	8 Core	States	1 Core	8 Core	
Anderson (6)	25	20	18,206,914	75	21	18,206,917
At (5)	36	29	31,999,395	118	33	31,999,440
At (6)	170	119	160,588,070	674	189	160,589,600
Bakery (7)	30	28	29,047,452	95	26	29,047,471
Bakery (8)	250	182	253,111,016	–	-	o.o.m.
Elevator (2)	30	23	11,428,766	74	21	11,428,767
Fisher (3)	10	9	2,896,705	12	3	2,896,705
Fisher (4)	7	7	1,272,254	5	1	1,272,254
Fisher (5)	126	86	101,027,986	541	141	101,028,339
Fisher (6)	16	13	8,321,728	37	10	8,321,728
Fisher (7)	555	360	386,281,613	-	-	o.o.m.
Frogs (4)	20	17	17,443,219	69	15	17,443,219
Frogs (5)	203	215	182,726,077	787	-	182,772,126
Lamport (8)	70	49	62,669,266	238	68	62,669,317
Mcs (5)	77	50	60,556,458	241	68	60,556,519
Mcs (6)	7	7	332,544	0	0	332,544
Philosophers (6)	36	27	14,348,901	122	36	14,348,906
Philosophers (7)	196	125	71,933,609	768	-	71,934,773
Philosophers (8)	105	70	43,046,407	405	-	43,046,720

taneously on a partitioned vector of successors. The comparison demonstrates only the influence to the whole Model Checking process; larger speed-ups were reached by considering only this aspect.

In order to compare CuDMoC with the current state-of-the-art in (multi-core) explicit-state Model Checking, additional experiments were done on the (most recent publicly available) releases of the DiVinE (version 2.2) and Spin (Holzmann, 2004) (version 5.2.4) model checker.

DiVinE instances were executed using the reachability command followed by a worker option `divine reachability -w N protocol.dve` with N denoting the number of cores to use and aborted when more then 11GB RAM were used. Table 8.4 shows the comparison in running time of the 1 core and the 8 core versions. Of course, DiVinE is not able to check some instances due to its exhaustive duplicate detection, it needs to store all visited states in full length, which is less memory efficient than Bitstate hashing. One interesting fact in the frogs (5) protocol is that DiVinE is only able to verify this instance in single-core mode. It has to be assumed that the queues, needed to perform communication between the cores consume too much memory. Additionally, the number of reached states is displayed, to indicate the number of states omitted. In the largest instance, the amount of states omitted is at most 3%. The speed-up averaged over all successful instances is 3,088 / 863 = 3.58 for one core and 632 / 484 = 1.31 for the 8 core implementation. DiVinE naturally utilizes all cores for

Table 8.5: Experimental results, comparing CUDA-driven model checker with Spin and Bitstate storage. Times given in seconds. Column Speed shows the quotient states/time. Protocol Mcs 5 was aborted after 10 hours, having generated 6,308,626.

	CuDMoC			Spin Bitstate BFS		
Protocol	1 Core [sec.]	Speed	States	1 Core [sec.]	Speed	States
Anderson (6)	25	728,276	18,206,914	26	698,282	18,155,353
Anderson (8)	669	804,923	538,493,685	228	618,216	140,953,300
At (5)	36	888,872	31,999,395	40	790,811	31,632,471
At (6)	170	944,635	160,588,070	146	727,404	106,201,110
Bakery (7)	30	968,248	29,047,452	29	942,202	27,323,870
Bakery (8)	250	1,012,444	253,111,016	156	78,283	12,212,250
Elevator (2)	30	380,958	11,428,766	19	601,239	11,423,554
Fisher (3)	10	289,670	2,896,705	4	724,170	2,896,681
Fisher (4)	7	181,750	1,272,254	2	636,131	1,272,262
Fisher (5)	126	801,809	101,027,986	141	614,026	86,577,752
Fisher (6)	16	520,108	8,321,728	13	639,997	8,319,972
Fisher (7)	555	696,002	386,281,613	242	547,841	132,577,710
Frogs (4)	20	872,160	17,443,219	19	916,191	17,407,634
Frogs (5)	203	900,128	182,726,077	136	853,619	116,092,290
Lamport (8)	70	895,275	62,669,266	8	917,817	7,342,543
Mcs (5)	77	786,447	60,556,458	–	–	0
Mcs (6)	7	47,506	332,544	1	36,598	36,598
Philosophers (6)	36	398,580	14,348,901	43	333,412	14,336,722
Philosophers (7)	196	367,008	71,933,609	229	297,427	68,110,830
Philosophers (8)	105	409,965	43,046,407	139	304,714	42,355,353

expansion, while CuDMoC uses the additional cores only for duplicate checking.

Spin is also able to manage an exhaustive representation of the *Closed* list, however, due to the memory limitations of an exhaustive search, the comparison of CuDMoC against Spin with the Bitstate implementation was chosen. Spin has two options for performing reachability, BFS and DFS. Table 8.5 presents the results in BFS, which has no multi-core implementation. Spin experiments were performed by calling `spin -a protocol.pm; cc -O3 -DSAFETY -DMEMLIM=12000 -DBITSTATE -DBFS -o pan.c;./pan -m10000000 -c0 -n -w28`. For the sake of clarity, also the number of reached states for both model checkers is presented. The number of states varies extremely for the larger instances. The explanation here is the diversity with the size of the Bitstate tables (in Spin 2^{28} = 268,435,456 entries were chosen, and a larger table could not be used because of the remaining memory that was occupied by the algorithm). *Speed* is used to denote the number generated states per second; CuDMoC achieves an average speed of 637,279 compared to Spin with an average speed of 593,598. Although the speed-up is not significant the fact should be highlighted that CuDMoC stores all the reached states on external memory for later usage, while these states are lost in Spin. Storing the information on external storage in Spin leads to a slowdown by a factor of 2 and more.

Table 8.6: Experimental results, comparing CUDA-driven model checker with SPIN and partial state storage. Times given in seconds. Speed denotes states per second.

SPIN Bitstate						
	1	Average	Generated	8	Average	Generated
Protocol	Core	Speed	States	Core	Speed	States
Anderson (6)	58	313,911	18,206,893	9	2,017,465	18,157,188
Anderson (8)	1316	275,800	362,954,000	78	1,859,341	145,028,600
At (5)	90	355,547	31,999,291	12	2,630,998	31,571,983
At (6)	399	339,403	135,422,110	42	2,476,482	104,012,280
Bakery (7)	48	573,577	27,531,713	8	3,413,837	27,310,696
Bakery (8)	456	488,071	222,560,800	39	3,062,315	119,430,320
Elevator (2)	47	243,165	11,428,769	8	1,427,956	11,423,654
Fisher (3)	7	413,815	2,896,707	2	1,448,344	2,896,689
Fisher (4)	2	636,128	1,272,256	1	1,272,298	1,272,298
Fisher (5)	275	367,375	101,028,340	36	2,397,127	86,296,605
Fisher (6)	20	416,086	8,321,730	4	2,079,982	8,319,929
Fisher (7)	1372	281,557	386,296,530	63	2,098,240	132,189,170
Frogs (4)	26	670,893	17,443,221	5	3,472,759	17,363,799
Frogs (5)	289	632,427	182,771,630	24	3,878,232	93,077,570
Lamport (8)	17	431,974	7,343,562	3	2,447,541	7,342,625
Mcs (5)	81	358,055	29,002,474	14	2,343,949	32,815,294
Mcs (6)	0	–	36,600	0	–	36,948
Philosophers (6)	26	387,130	10,065,395	17	843,330	14,336,624
Philosophers (7)	351	183,494	64,406,569	51	1,217,002	62,067,145
Philosophers (8)	12	766,795	9,201,551	35	1,043,143	36,510,039

As the SPIN BFS algorithm is not parallelizable, the algorithm is compared to the DFS version and Bitstate hashing called via `spin -a protocol.pm; cc -O3 -DSAFETY -DMEMLIM=8000 -DBITSTATE -DNCORE=N -DNSUCC -DVMAX=144 -o pan.c;./pan -m10000000 -c0 -n -w27` (using 1 core), and `-w25` (using 8 cores) with N denoting the number of cores. Table 8.6 shows the running times and per node efficiencies for the tested protocols. Since the numbers for the 1 core CUDMOC implementation are identical in Table 8.5, here only the values for the 8 core implementation are presented. The 8 core implementation of the DFS algorithm is always faster then the CUDMOC implementation. A closer inspection of the number of the visited states reveals that the number of cores has an impact on the size of the Bitstate table, thus resulting in different amounts of visited states. In the Anderson (8) protocol, which is the largest checked protocol, CUDMOC identifies 538,493,685 unique states, while the SPIN 8 core implementation reaches only 145,028,600 states, omitting nearly 70% of the state space. Additional observations showed that at the beginning of the search the speed is higher, since new states are reached more often, than at the end, where a large amount of reached states has already been explored.

Table 8.7: Comparing GPU- with CPU-based Performances (The CPU instance of Peg-Solitaire has been stopped in BFS-Layer 17, o.o.m. denotes out of memory)

Protocol	Runtime in hh:mm			
	DIVINE	SPIN	CPU	GPU
Telephony (6)	o.o.m.	o.o.m.	4:42	3:03
Telephony (7)	0:01	0:00.5	0:04	0:02
Telephony (8)	o.o.m.	o.o.m.	2:22	1:09
Szymanski (5)	0:03	0:01	0:12	0:08
Anderson (8)	o.o.m.	o.o.m.	1:32	0:47
At (7)	o.o.m.	o.o.m.	1:56	0:45
Peg-Solitaire (6)	o.o.m.	o.o.m.	o.o.t.	14:57
(first 17 layers)			11:52	1:20

Table 8.8: State space sizes of various protocols.

Protocol	State Space (in GB)		
	States	uncompressed	compressed
Telephony (6)	1,495,154,914	69.0	12
Telephony (7)	21,960,309	1.1	0.168
Telephony (8)	854,245,188	43.0	6.4
Szymanski (5)	79,518,741	3.8	0.6
Anderson (8)	538,699,094	26.0	4.1
At (7)	819,243,858	34.0	6.2
Peg-Solitaire (6)	2,383,981,575	134.0	18
(first 17 layers)	246,328,560	13.8	1.8

8.2.2 Experiments with Delayed Duplicate Detection

For comparing delayed duplicate detection strategies, different sorting strategies were evaluated: the system built-in CPU QUICKSORT implementation, the GPU QUICKSORT implementation of (Cederman and Tsigas, 2008) and a BITONIC SORT routine[4], all adapted to sort state vectors instead of numbers. In the end, BITONIC SORT was adopted and hash partitioning as well as state compression to 64-bit as motivated above.

Overall runtime comparison

Table 8.7 displays the total run-times of the model checker subject to CPU- and GPU-based state space exploration on disk for the selected benchmarks protocols. Using the GPU induces the model checker to perform consistently better. To get CPU data in a feasible amount of time, an experiment terminating Peg-Solitaire (6) after layer 17 was

[4] Used sources available at http://courses.ece.illinois.edu/ece498/al/HallOfFame.html

Table 8.9: Comparing GPU- with CPU-based Performances in differential stages.

Protocol	Telephony			Szymanski	Anderson	At	Peg-Solitaire (17 layers)
Instance	(6)	(7)	(8)	(5)	(8)	(7)	(6)
Enabling Transitions							
CPU	3,654s	59s	2,362s	188s	720s	1,727s	32,226s
GPU	115s	1s	78s	5s	24s	46s	429s
Speedup	31.7	59.0	30.2	37.6	30.0	37.5	75.6
Generating Successors							
CPU	1,964s	28s	1,193s	74s	734s	801s	4,088s
GPU	301s	4s	196s	12s	121s	140s	448s
Speedup	6.5	7.0	6.1	6.2	6.0	5.7	9.1
Sorting							
CPU	4,372s	62s	2,447s	192s	1,585s	2,002s	3,220s
GPU	180s	41s	134s	82s	153s	86s	129s
Speedup	24.3	1.51	18.3	2.4	10.4	23.2	25.0

performed, when it had generated about 10% of all unique states and the results are shown in the last line. While the state space of the At (7) protocol is larger than that of the partially generated Peg-Solitaire instance, surprisingly, the total time for generating it on the CPU is smaller. This is due to the fact that the out degree of the Peg-Solitaire protocol is much higher and 90% of the generated successors are duplicates which are discarded. For the sake of completeness a comparison of the algorithm with the DiVinE-MC implementation and Spin binaries was undertaken. Since DiVinE-MC and Spin are only able to check instances that fit into RAM (both were allowed to use 12 GB), we see that they are not terminating on most models. If they do they are much faster, since both checkers use hashing for state storage, which is very fast compared to an implementation that uses sorting-based delayed duplicate detection for an increased external-memory performance. Table 8.8 shows the various state space sizes of the protocols.

Individual runtimes of the three stages

The individual speed-ups for enabling transitions, successor generation and sorting are depicted in Table 8.9 showing the protocol and its checked property in the first row. Remaining rows are divided according to the stages. The timing information is the sum of the efforts for all BFS-Layers in the state space generating process. The table strongly suggests that the GPU should be used to perform similar tasks on all threads. It also identifies the impact of the GPU being larger for enabling the transitions than for generating the successors. This is due to the fact that the task of checking a transition is equal for all threads in one group and run simultaneously. When generating a state, each thread applies a transition according to its index. In the worst case each

Table 8.10: Comparison of CPU and GPU times for the distinct stages on the first 17 BFS-Layers of the Peg-Solitaire (6) protocol. (The CPU experiment was stopped due to obvious suboptimal performance.)

	Times[s]		
Operation	CPU	GPU	Ratio
Reading Search Frontier States from HDD	397s	402s	
Find active Transitions (on the GPU incl. Transfer)	32,226s	429s	75.11
Applying Transitions (on the GPU incl. Transfer)	4,088s	448s	9.13
Compressing States (Hash Function and Bucketing)	877s	1,488s	
Sorting Compressed States	3,220s	129s	24.96
Subtracting Previous Layers Read from HDD	1,538s	1,577s	
Writing Duplicate-Free Layer File to HDD	29s	45s	
Appending Full States to Search Frontier on HDD	146s	160s	
Other memory operations	178s	167s	
Total Time	42,699s	4,845s	9.61

thread applies a different transition, reducing the amount of parallelism in memory access. The last part of the table shows a sorting speed-up that differs widely between instances. This was a surprising result, since the work of sorting is the same on all instances, where a constant number of buckets (VRAM/SRAM) with an in average constant number of elements (SRAM/64/8/2) is sorted. Looking carefully at the state space can clarify why the speed-up differs. Since the maximal BFS depth varies, and the size of each individual BFS-Layer is different, sorting is not a unified task. The small speed-up of the Szymanski (5) instance is explained by many small layers, and, for each layer, all buckets have to be copied to the GPU.

Finally, a profiling experiment was performed to uncover remaining performance bottlenecks. A detailed profile for the Peg-Solitaire (6) Protocol (explored up to BFS-Level 17) is provided in Table 8.10. Most of the time is lost in pre- and post-processing the data. The term that harms most is due to the subtraction of previous layers, for which strategies like revisiting resistance (Barnat *et al.*, 2008b) and layered duplicate detection (Lamborn and Hansen, 2008) should apply. Using multiple external drives would also reduce the impact of reading and writing and yielding a better factor.

8.3 Summary

This chapter evaluates the GPUSSD framework on the minimal-counterexample implementation and the GPU supported model checker CuDMoC. Having shown good results in using a perfect hash function to compress the state space the PHF is externalized to SSDs and show a significant performance boost compared to HDDs. For a faster state space generation the search is partitioned into three stages and ported to the GPU applying the successor pointing strategy. Duplicate detection can be either semi-immediate providing fast checking but being limited in by the RAM, or delayed imposing longer search times but enabling to examine larger state space graphs.

Part III

Action Planning

Chapter 9

Introduction to Action Planning

There is no doubt that the success of planners is sensitive to the amount of computational resources available. It is not hard to predict that due to economic pressure parallel computing on an increased number of cores both in central processing and graphics processing units will be essential to solve challenging problems in the future.

Motivated by the results in Explicit Model Checking the proposed strategies will be applied to Action Planning. In contrast to Edelkamp *et al.* (2010a) where a software implementation is translated into a planning problem and solved by an existing planner, this approach proposes a GPU extended planner. Due to the similarities in both domains some decisions undertaken in the previous part can be inherited. However, the increased number of transitions, and larger states connected with heuristic search, requested for new strategies.

Unfortunately, beside the published results on the planner developed during this work (Sulewski *et al.*, 2011), so far no domain-independent planner has been proposed that utilizes the GPU. This is partly due to the fact that the single instruction multiple data architecture of GPUs is more closely related to a vector computer that induces a distinguished programming model.

Structure of the chapter: Having introduced planning and defined the necessary properties for domain, action and problem descriptions this chapter draws a connection to the state space search. As a visual example of the Planning Domain Definition Language the thesis problem from the introduction is given followed by related work in this discipline.

9.1 Modeling of Planning Problems

The Artificial Intelligence classifies *planning* as finding a sequence of actions that transform a given initial state into a goal state satisfying certain conditions. These conditions are given additionally to a *domain* description defining the available actions to the agent. Having generated the initial state actions are applied implicitly constructing a graph whose states are checked for fulfilling the goal conditions. The following definitions are based on STRIPS planning introduced by Fikes and Nilsson (1971). When a goal state is found the path from the initial to the goal state is used as the sequence representing the *plan*. A *planning problem* is defined as follows:

Definition 29 (Planning Problem) *A classical* planning problem *is a tuple* $\mathcal{P} = (\mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G})$ *with*

- $\mathcal{F}$ *being a set of fluents,*
- $\mathcal{A}$ *a set of actions,*
- $\mathcal{I} \subseteq \mathcal{F}$ *the fluents that hold in the initial state and*
- $\mathcal{G} \subseteq \mathcal{F}$ *the fluents that need to be satisfied in any goal state.*

where $\hat{s}$ is build up of $\mathcal{I}$ and $\neg\mathcal{I}$ the fluents that do not hold in the initial state.

Actions in planning can be mapped to edges in a graph and seen as transitions from a parent to a successor. A slight variation exists to the transition Definition 6 on page 4 since the postconditions in planning are divided into two parts the *add* and the *delete* effects. Additionally, actions can have costs making a Dijkstra search unavoidable.

Definition 30 (Action) *An* action $a \in \mathcal{A}$ *is a tuple* $a = (P, A, D)$*, with*

- $P \subseteq \mathcal{F}$ *the* precondition *that needs to be satisfied so that action* a *can be applied,*
- $A \subseteq \mathcal{F}$ *the set of fluents added to the current state also called* add *effects and*
- $D \subseteq \mathcal{F}$ *the set of fluents removed from it after applying the action often denoted as* delete *effects.*

The aim is to find a path called *plan*, i. e., a sequence of actions, that transforms the initial state into a goal state. In case of optimal planning, this plan must be minimal in terms of path length or path cost.

In *cost-based planning*, actions can be assigned certain costs, according so that the planning problem is extended to the tuple $\mathcal{P}_c = (\mathcal{F}, \mathcal{A}, cost, \mathcal{I}, \mathcal{G})$ with $cost : \mathcal{A} \mapsto \mathbb{N}_0^+$. For such a problem, the total cost of the resulting plan is the sum of the costs of all actions within the plan and in case of optimal planning, the plan with minimal total cost has to be found.

Oversubscription planning is the extension of classical planning to so-called soft goals, i. e., goals that may be satisfied but are not obligatory. Thus, the problem is a tuple $\mathcal{P}_o = (\mathcal{F}, \mathcal{A}, \mathcal{I}, \mathcal{G}, \mathit{utility})$ with $\mathit{utility} : 2^{\mathcal{F}} \mapsto \mathbb{N}$ a function that assigns a certain reward for achieving a soft goal. The utility of the resulting plan is the utility of the achieved goal state and the aim is to maximize it.

Finally, *net-benefit planning* contains action costs and soft goals, i. e., the net-benefit planning problem is a tuple $\mathcal{P}_{nb} = (\mathcal{F}, \mathcal{A}, cost, \mathcal{I}, \mathcal{G}, \mathit{utility})$. The net-benefit of a plan is the utility achieved by the plan minus the total action cost needed to achieve it and in case of optimal planning one is interested in finding a plan that maximizes this net-benefit.

Nowadays, the *Planning Domain Definition Language* (PDDL) (McDermott, 1998) is the most frequently used formalism for modeling. It supports, among others, complex Boolean formulas for the preconditions of actions and goal descriptions, numerical state variables, and rational action costs (which are scaled to integers). A PDDL domain in which all predicates are atoms and in which all actions have zero parameters is *grounded* (Kissmann and Edelkamp, 2009). Matching the formalization above, this planner assumes a grounded problem representation with a fully instantiated PDDL description as input (e. g., provided by Hoffmann's *adl2strips*, Helmert's *translate*, or Haslum's *pddlcat*).

9.2 PDDL Example of the Thesis Problem

Figures 9.1 and 9.2 visualize the thesis Example 1.1 on page 7 in the introduction using PDDL. After defining the name of the `domain` the requirements are formed. Each variable is handled as an `object` but the `typing` requirements permit the usage of derived user defined types for an easier distinction of them. This example uses four types, `student` and `friend` for the student and the friends, `mode` for the mode a student or a friend can be in and `feeling` for the properties of the student. In the following the variable `completeness` is defined to describe the completeness of the document and followed by the actions.

Each action starts with the tag `:action` followed by a name for it. Then the used parameters are depicted after the tag `:parameters` and classified in a type class. The middle part of the action prefixed with the tag `:precondition` describes the preconditions being checked before the effects, given in the third part triggered with `:effect`. In PDDL the effects are given as one boolean formula and all the delete effects are concatenated and prefixed with a `not` statement. Having defined the domain the `problem` is described starting with the presentation of the objects, here one student and one friend and the definition of modes. After defining the enabled (and disabled) fluents the `goal` condition is given. Since no constraint on the number of applicable actions is given in planning all preconditions present in the definition have to be checked in each state.

9.3 Related Work

The approach is distinct to existing multi-core approaches: Multiple cores on the CPU are only used for delayed duplicate detection. In contrast, other parallelizations distribute the search space based on different sorts of hash-partitioning like HDA* (Kishimoto *et al.*, 2009). Parallel version of A* (Evett *et al.*, 1995) and transposition-driven scheduling (Romein *et al.*, 1999) exploit parallelism in the search space, while PBNF (Burns *et al.*, 2009b) extends the idea of parallel structured duplicate detection (Zhou and Hansen, 2007), exploiting the locality of a search space. Planners trying to catch up with these hardware trends enhance state space planning using, e. g., different multi-core CPU approaches (Vidal *et al.*, 2010; Burns *et al.*, 2009b).

Arvand (Nakhost *et al.*, 2010) is a planner based on Monte-Carlo searches and

```
(define (domain thesis-writing)
  (:requirements :typing :numeric-fluents)
  (:types student friend - object
          mode - object
          feeling - object)
(:predicates
  (inmodes ?s - student ?m - mode)
  (inmodef ?f - friend ?m - mode)
  (feel ?s - student ?f- feeling ))
(:functions (completeness) - number)
(:objects
  (student1 - student)
  (friend1 - friend)
  (sleeping eating thinking writing enjoyingTime reading -mode)
  (sleepy hungry - feeling))


(:action sleepToEat
  :parameters (?student - student)
  :precondition (inmodes ?student sleeping)
  :effect (and (inmodes ?student eating)
               (not (feel ?student sleepy))))
(:action eatToThink
  :parameters (?student - student)
  :precondition inmodes (?student eating)
  :effect (and (inmodes ?student thinking)
               (not (feel ?student hungry))))
(:action thinkToWrite
  :parameters (?student - student)
  :precondition (inmodes ?student thinking)
  :effect (inmodes ?student writing))
(:action writeToSleep
  :parameters (?student - student)
  :precondition (and (inmodes ?student writing)
                     (feel ?student sleepy))
  :effect (and (inmodes ?student sleeping)
               (feel ?student hungry)
               (increase (completeness) 10)))
(:action writeToEat
  :parameters (?student - student)
  :precondition (and (inmodes ?student writing)
                     (feel ?student hungry))
  :effect (and (inmodes ?student eating)
               (feel ?student sleepy)
               (increase (completeness) 10)))
```

Figure 9.1: PDDL language example describing Figure 1.1 on Page 7 (First part).

```
(:action writeToThink
  :parameters (?student - student)
  :precondition (and (inmodes ?student writing)
                     (not (feel ?student hungry)
      (feel ?student sleepy)))
  :effect (and (inmodes ?student thinking)
               (feel ?student hungry)
               (increase (completeness) 10)))

(:action proofRead
  :parameters (?friend - friend)
  :precondition (and (inmodef ?friend enjoyingTime)
                     (> ?completeness 0))
  :effect (inmodef ?friend reading))
(:action corrected
  :parameters (?friend - freind)
  :precondition (inmodef ?friend reading)
  :effect (and (inmodef ?friend enjoingTime)
               (decrease (completeness) 1)))
)

(define (problem finishing_thesis_possible)
  (:domain thesis-writing)

(:init
    (feel student1 sleepy) (feel student1 hungry)
    (inmode student1 sleeping) (inmode friend1 enjoyingTime)
    (= (completeness) 0)
)
  (:goal
    (>= (completeness) 100)
  )
)
```

Figure 9.2: PDDL language example describing Figure 1.1 on Page 7 (Second part).

random restarts. As different starts are independent, assuming sufficient memory these searches can easily be parallelized. The results are promising but solutions are typically sub-optimal. (Multi-core) UCT a dynamical tree-growing learning algorithm on top of Monte-Carlo search, can be used for finding optimal solutions based on lock-free hashing (Enzenberger and Müller, 2009), but has not yet been implemented.

Often, suboptimal planning is addressed (Burns *et al.*, 2009a; Nakhost *et al.*, 2010), but also a sizable number of optimal parallel planners has been developed in the last few years (e. g., Zhou and Hansen (2007) and Zhou *et al.* (2010).

External approaches, like proposed by Edelkamp and Jabbar (2006a) and Edelkamp *et al.* (2006) utilize magnetic media devices to store the state space on. While this approaches also enable unlimited storage capabilities they do not profit from the RAM since it is just used as a buffer.

9.4 Summary

This chapter introduced Action Planning and the PDDL language used to describe problems for a planner. An example is given according to the example in the introduction and aspects of modeling a problem are discussed. The following section provides an overview on the related work in Planning. The following chapters will propose a planner enhanced by a GPU.

Chapter 10

Action Planning on the GPU

This chapter proposes a domain-independent CUDA driven planner (CUDPLAN) for which precondition checks and successor generation are executed on the GPU published in (Sulewski *et al.*, 2011). As GPUs usually have no cache hierarchy and are relatively slow in accessing the global memory on the graphics card, duplicate detection is executed in the RAM using the CPU. For large state spaces the planner supports exploration on disk, together with either delayed duplicate detection (Korf, 2008a) or Bitstate hash tables (Bloom, 1970). As the main interest is optimal planning, a so-called *lock-free* hash table is used, a promising data structure based on low-level compare-and-swap (CAS) operations that avoids using variables for locking regions of the memory exlusively to an unique thread (Laarman *et al.*, 2010; Enzenberger and Müller, 2009).

Structure of the chapter: This chapter begins with an argumentation on the chosen strategies for the framework followed by details on the realization of each strategy. After motivating the used duplicate detection technique an eager planning algorithm is presented partitioning the planning problem according to the specifications given in the framework for GPU state space search.

10.1 Strategies from the GPUSSD-BFS Framework

When applying the proposed framework to port a search problem to support the GPU the strategies to chose have to be evaluated. The decision for the generation stages is presented in the next section followed by a detailed description of the algorithm.

10.1.1 Successor Generation on the GPU

Since the preconditions and effects of each action will be transferred to the Reverse Polish Notation as proposed in Part I, the first decision to be taken is the action evaluation and successor generation strategy. The preconditions of actions can become arbitrarily complex in a planning problem, so a repetition of the action evaluation should be omitted when possible. Thus the strategy to choose is the *successor pointing strategy* with a bit vector having mapped the indexes of each bit to an action according to the problem description. Due to the possibly large amount of actions an advantage of this approach is to separate the preconditions from the effects increasing the probability to store each partition in the SRAM of the GPU. However, due to the number of actions being very large the size of the bit vector can easily become larger than the state.

As an example of the amount of actions, consider a state with only two Boolean variables `a,b`. The possible preconditions for all actions are (a), (b), $(\bar{a})$, $(\bar{b})$, (ab), $(\overline{ab})$, $(\bar{a}b)$, $(a\bar{b})$ resulting in an enabled bit vector of length 8 bits for a state-vector of length 2 bits. A compression method is realized easily here, by noticing that an enabled action with precondition (ab) also imposes (a) and (b), reducing the number of necessary bits to 4. When the remaining preconditions ((ab), $(\overline{ab})$, $(\bar{a}b)$, $(a\bar{b})$) are analyzed further it is obvious that only one of it can be enabled at a time, such an SAS$^+$ representation reduces the space to 2 bits by storing the index of the enabled action in binary representation.

10.2 GPU Planning Algorithm

As the main interest in this work is optimal planning, approximate hashing cannot be used. Therefore the duplicate detection to be used is lock-less hashing combined with an $Open$ list stored externally. A sorting based strategy as utilized in Model Checking is not efficient in planning due to the larger size of the states. Even using compression methods each state utilizes several bytes in larger instances making an effective hash based sorting approach impossible.

10.2.1 Planner Architecture

The implemented planner utilizes two different kernels, one designed to generate successors for cost-optimal and one that deals with optimal planning for oversubscribed and net-benefit planning problems. Both kernels are able to deal with numbers. The main difference of the two kernels is that the former stops at expanding the first goal node, while the latter overwrites the action after being executed with the metric value of the state.

For successor generation on the GPU the satisfaction of the preconditions of actions against the state set that has been copied to the GPU is checked and the effects to the ones that have passed the test are applied. As this is a considerable amount of work for each GPU core, the postfix representation of the propositional and numerical expressions that appear in the precondition and effects introduced in Section 4.5 is exploited. These representations are precomputed and broadcasted in the GPU. Extensive tests showed that using the postfix representation enhances checking the validity of a precondition and the computation of the assignment to an effect variable on the GPU due to using a flat evaluation stack.

Conceptually (and as illustrated in the simplified pseudo-code in Algorithm 10.2), for each track one kernel exists for parallel state expansion on the GPU. In the practical implementation, however, successors are generated in two stages on the GPU according to Algorithm 4.1 (page 64). In a first step, the preconditions of the actions are checked against the states in the GPU. Rather complex Boolean and numerical expressions are allowed and the preconditions of all actions are checked against the state (without applying any static filter).

In domains without (or with uniform) action costs, a breadth-first enumeration of the search space is sufficient, while in domains with action costs optimal path finding resorts to exploration of paths with minimal cost. If the action costs are integers (or the used costs can be transformed as integers), a cost-based implementation of Dijkstra's shortest path algorithm on buckets is possible (Dijkstra, 1959; Dial, 1969).

Subsequently, with a *buffer-filling* implementation variant, the *eager* state expansion on the GPU is preferred by means that states may be expanded earlier than dictated by the monotonic non-decreasing ordering of the costs. For this case the first expanded goal state no longer necessarily has optimal solution cost. However, given that costs are non-negative, states can be omitted if the current cost exceeds the best one found so far. The performance gains due to improved parallel processing can relativize the additional amount for expanding more states than necessary.

Moreover, duplicate elimination is relaxed. While the algorithm prevents inserting states into $Open$ with larger costs than the one stored in the hash table, inserting states with smaller costs than the ones stored in the hash table does not imply that the latter ones are removed. This can result in significant re-expansions.

The pseudo-code of the eager buffer-filling planning algorithm is shown in Algorithm 10.1 with an expansion kernel routine for the GPU sketched in Algorithm 10.2.[1] $Open$ is implemented as a hash map of lists (with the costs being the keys). This sparse representation of a virtual bucket array of lists allows to deal with arbitrarily large action costs. $Closed$ is implemented as a (lock-free) hash table. All other structures are implemented as vectors or lists.

The search is divided into stages like suggested by the framework and extended by a *termination check* to stop when an optimal plan is found. The first stage differs only marginally from the proposed successor pointing strategy by an extension to check all available $Open$ lists for states. After all available states are collected stage 2 begins which is extended to support action costs while generating the successors. Here, the tuple (t, s) consisting of a transition id and the appropriate state as proposed in the generation step for the successor pointing strategy is extended to a triple $(t, s, cost)$ enabling the GPU to compute the costs of the generated successor. Naturally this stage is also extended to support a set of $Open$ lists rather than one list.

A major modification was necessary to the third stage of the framework. While the task to check for the existence of a duplicate in $Closed$ remains, a state is first checked for being a goal state. Having done this a state s is only inserted to $Open$ if $cost(s)$ is lower than $best$, the costs of the current best solution or until no solution is found ∞.

For the sake of clarity, the code abstracts from the implementation. While $Open$ is stored externally as a number of files and Locking and Unlocking is done internally on pointers to these files, $Closed$ is a lock-free hashing table enabling internal duplicate detection based just on the hash value.

[1] In the algorithms square brackets are used to denote concepts only relevant in net-benefit planning.

Algorithm 10.1: Optimal Eager Buffer-Filling GPU Planning algorithm

```
Input: $\mathcal{P}_{c[nb]} = (\mathcal{F}, \mathcal{A}, cost, \hat{s}, \mathcal{G}[, utility])$
Output: Optimal [value of utility minus] action cost
$best \leftarrow \infty$ ;                                  {initialize best to infinite}
$i \leftarrow 0$ ;                                          {reset the actual cost indicator}
$Open_0 \leftarrow \hat{s}$ ;                               {store $\hat{s}$ in $Open$ list indexed with 0 costs}
$Closed \leftarrow (\hat{s}, 0)$ ;    {clear $Closed$ list and store $\hat{s}$ with appropriate costs}
while true do
    Stage 1 - Generate sets of active transitions
    $Active \leftarrow \emptyset$ ;
    while $|\text{Active}| \neq |Open_{i \cdots max}|$ do    {until all frontier states are processed}
        while $(\exists j \geq i : Open_j \neq \emptyset)$ do    {while $s$ in $Open$ with costs $\geq i$ exist}
            fillVRAM($s \in Open_j$) ;                      {copy states to VRAM}
        $Active \leftarrow Active \cup$ GPU-Kernel Determine Transitions() ;
    Stage 2 - Generate sets of successors and compute costs
    $Successors \leftarrow \emptyset$ ;                     {will contain pairs of $(s, cost\ (s))$}
    while Active $\neq \emptyset$ do                        {Until all actions processed}
        while $(\exists j \geq i : Open_j \neq \emptyset)$ do    {while $s$ in $Open$ with costs $\geq i$ exist}
            fillVRAM($\{(t \in Active, s \in Open_j, j), \cdots\}$)
            {Move triples of an active transition, a state and its costs to VRAM}
            $Open_j \leftarrow Open_j \cap$ VRAM ;          {remove states from $Open$ }
            $Active \leftarrow Active \cap$ VRAM ;          {remove states from $Active$}
        $Successors \leftarrow Successors \cup$ GPU-Kernel Generate Successors();
    Stage 3 - Check for goal, remove duplicates and rebuild $Open$
    for $(s, c) \in$ Successors do
        if $(\mathcal{G} \subseteq s)$ then                 {if state is a goal record its plan cost}
            $best \leftarrow \min\{best, [utility(s) - c]\}$ ;
            if $\neg utility$ then return $best$ ;
        $c' \leftarrow (cost(Search(s)$ in $Closed) \vee \infty)$ ;
        {check $Closed$ for costs of a duplicate}
        if $(c < c')$ then                                  {plan with lower costs found}
            $Closed \leftarrow (s, c)$ ;
            if $(c < best)$ then {only if plan costs are $< best$ add state to $Open$ }
                Lock $Open_c$ ;
                {$Open$ is locked on $cost$ level to enable parallelism}
                $Open_c \leftarrow s$ ;
                Unlock $Open_c$ ;
                                                            {Termination check}
    if $\forall j \geq i : Open_j = \emptyset$ then         {all $Open$ lists with costs $\geq i$ are empty}
        if best $= \infty$ then return $unsolvable$ ;       {plan not found}
        else return $best$                                  {return cost of found optimal plan}
    $i \leftarrow \min\{j \mid j > i \wedge Open_j \neq \emptyset\}$ ;    {increase $i$ to next costs layer}
```

Algorithm 10.2: *GPU-Kernel* Generate Successors in Planning

```
Input: $\{(t \in \mathit{Active}, s \in Open_j, j), \cdots\})$ a set of triples with an active
    transition, the appropriate state and its costs
Output: $\{(s_s \in \mathit{Successors}, \mathit{cost}(s_s)), \cdots\}$ pairs of a successor and its costs
for each group $g$ do in parallel    {partially distributed computation}
    for each thread $p : 0 \leq p < d$ do in parallel    {distributed computation}
        $\mathit{Successors} \leftarrow \mathit{Successors} \cup (\mathit{Successors}(s_{g\cdot d+p}, t_{t_{g\cdot d+p}}), c + \mathit{cost}(a))$ ;
        {add successor and its costs}
return Successors;
```

Considering net-benefit planning, the net-benefit is computed as the utility for each soft goal established minus the label of the cost layer (line 18). The continuously improving bound *best* records the best net-benefit and terminates the exploration in case the optimum has been proven. As a result (and as a side effect of the eager buffer-filling approach), net-benefit problems can also be solved optimally with Algorithm 10.1.

Theorem 2 *(Optimality) For cost-based planning problems $\mathcal{P}_c$ and net-benefit planning problems $\mathcal{P}_{nb}$ the eager buffer-filling planning algorithm on the GPU computes the costs of an optimal solution.*

Proof. States are only eliminated from the $Open$ list in case a strictly better (in terms of accumulated action costs) matching state has been found in $Closed$, or if the non-decreasing accumulated metric value (measured either in accumulated action cost or net-benefit) is worse than the currently best established goal metric value (measured either in accumulated action cost or net-benefit). □

Theorem 3 *(Efficiency) For cost-based optimal planning problems $\mathcal{P}_c$ and net-benefit planning problems $\mathcal{P}_{nb}$ the eager buffer-filling planning algorithm expands each state at most once for every action cost-layer of the search frontier.*

Proof. As in each cost-layer a full duplicate detection is applied, any state can appear at most once per layer. □

As the number of re-expansions depends on the domain, in the experiments a version that avoids buffer-filling and performs a Dijkstra-like state-space traversal was evaluated. The changes for the pseudo-code are moderate. For cost-based planning it stops at the first goal to be expanded.

10.3 Summary

This chapter applied the framework proposed in Part I to design a GPU enhanced planner. The process of planning, aka state space searching was divided into three stages, an action evaluation and a successor generation strategy was chosen. The preconditions and effects are transferred to the Reverse Polish Notation to be efficiently evaluated on the GPU. The next chapter will evaluate this proposal on problems from the international planning competition 2008.

Chapter 11

Experimental evaluation

The benchmark domains are taken from the sequential optimal and the optimal net-benefit tracks of the 2008 international planning competition (IPC).[1] The competitors of CUDPLAN are the two best planners in each track: For sequential optimal planning these are the baseline planner (an explicit-state planner with A* and zero-heuristic) and Gamer (Edelkamp and Kissmann, 2009) (a BDD-based bidirectional cost-first search planner). For optimal net-benefit they are Gamer (featuring BDDs and unidirectional cost-first branch-and-bound planning) and MIPS-XXL (Edelkamp and Jabbar, 2008) (an explicit-state breadth-first external-memory planner).

The computer infrastructure is the 64-bit system introduced in Chapter 2. All experiments are canceled after exhausting memory or 15 minutes of wall-clock time.[2] For lock-free hashing, our implementation uses the GNU gcc compiler for 64-bit x86 target platforms. A gcc built-in is used for the compare-and-swap operation and reads and writes from and to buckets are marked volatile.

Structure of the chapter: In the next section results on the evaluation are presented and discussed. Several test series are presented comparing the GPU planner (CUDPLAN) to state of the art planners on a recent system.

11.1 Results of the Evaluation

In Figures 11.1 to 11.14 the running time in seconds for each instance of a domain if it was solved by a planner is plotted. The planners are scored by the same system as in the IPC 2008. For each domain the number of solved instances is accumulated. If a domain is available in different formulations, e. g., the elevators domain in the net-benefit track, the maximum number of solved instances over all these formulations is used.

[1] `http://ipc.informatik.uni-freiburg.de`

[2] For the version of Gamer used for the problems from the sequential-optimal track, we set the time for the backward search to 450 seconds.

Figures 11.1 to 11.6 display the results obtained in the six benchmarks of the optimal net-benefit track of ICP-6 while Plots 11.7 to 11.14 deliver the results of the sequential optimal track. Here the task is, analogue to classical planning, to find a plan with minimal length or, if costs are given, with minimal costs.

As can be concluded from the plots the GPU planner with buffer-filling enabled can solve 43 problems, the GPU planner without buffer-filling 52, MIPS-XXL 22 and Gamer the highest number of 57 problems. However, investigating the cause for unsolved instances can be reported that while Gamer was killed due to reaching the limit of 15 minutes, the GPU planner without buffer-filling never reached this timeout. In many of the instances the lock-free hash table was completely filled and the planner stopped. Thus, a larger amount of main memory would turn the picture in favor of the GPU planner.

The plots also depict another fact about the GPU planner, i. e., it depends on hard problems. All plots but the Crewplanning domain reveal that there is a threshold where the GPU planner is significantly faster than the other planners. Furthermore, in Crewplanning buffer-filling outperforms the algorithm without buffer-filling while in the others this strategy is less effective. The most likely reason for this is the fact that the state space of this domain is rather *flat*, i. e., each cost layer is small. A flat state space means a better distribution of states into separate layers and thus fewer re-expansions.

Numbers of expanded nodes were not compared due to the basic differences of the planners. MIPS-XXL, being developed to solve large instances using external memory, and Gamer, executing a symbolic search, are not comparable in this respect with the explicit state internal memory GPU-based planner.

Figures 11.1 to 11.6 display results obtained in the eight *STanford Research Institute Problem Solver* (STRIPS) benchmark of the sequential optimal track of IPC-6.

The GPU planner with buffer-filling solves 124 problems, the GPU planner without buffer-filling 138 problems, the baseline planner 134 problems and Gamer 127 problems. The advantage used by the GPU planner here is the large amount of Boolean fluents in the problem description. Due to the usage of only one bit per fluent, compared to 32-bits for integer fluents, the vector identifying the state is smaller and more vectors fit into the hash table, enabling the planner to examine larger state spaces and better utilizing the computation power of the GPU.

The plots emphasize the advantages of using the GPU. In every domain a threshold exists where the GPU planner is faster then the CPU based baseline planner. Unfortunately, even 24 GB of RAM are not enough to solve the hardest problems and demonstrate the achievable speed. In the best cases the speedup factor, defined as the running time of baseline divided by the running time of the GPU planner, exceeds a factor of seven, rising with the size of the problem.

The GPU planner with buffer-filling behaves only satisfactorily in the Parcprinter domain, being slower than the other planners on all the remaining ones. Here also a flat state space exists, in this case due to the strong divergence of the costs, keeping the number of re-expansions small.

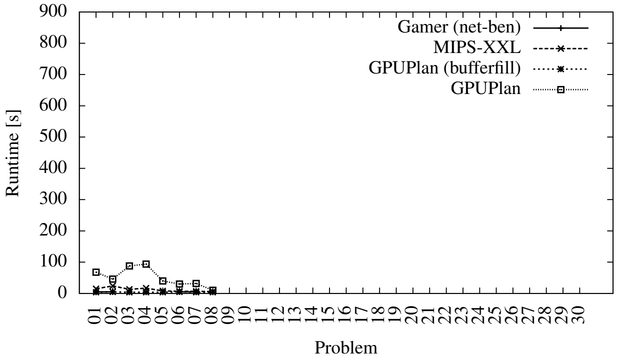


Figure 11.1: Optimal Net-Benefit Track - Crewplanning runtimes.

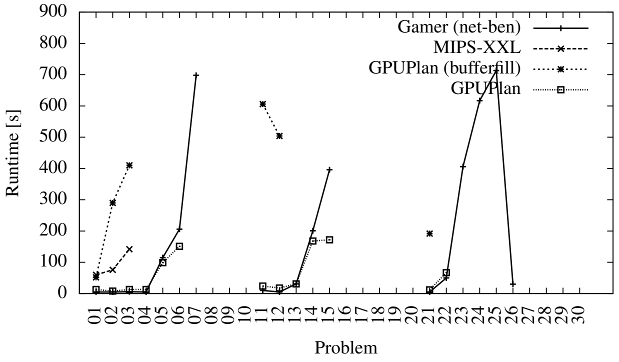


Figure 11.2: Optimal Net-Benefit Track - Numeric Elevators runtimes.

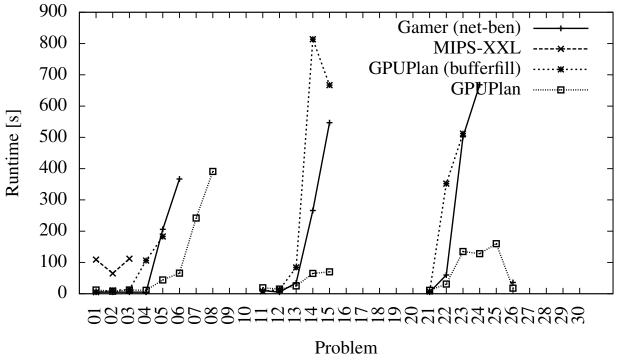


Figure 11.3: Optimal Net-Benefit Track - STRIPS Elevators runtimes.

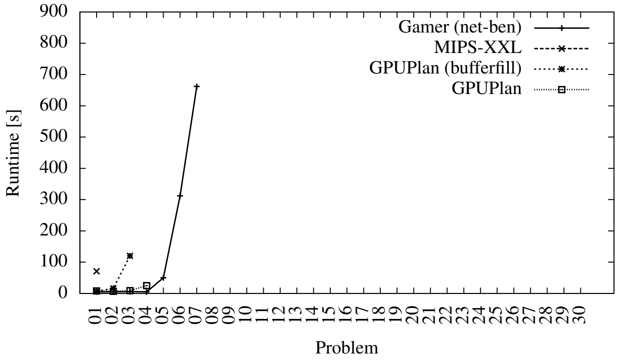


Figure 11.4: Optimal Net-Benefit Track - Openstacks runtimes.

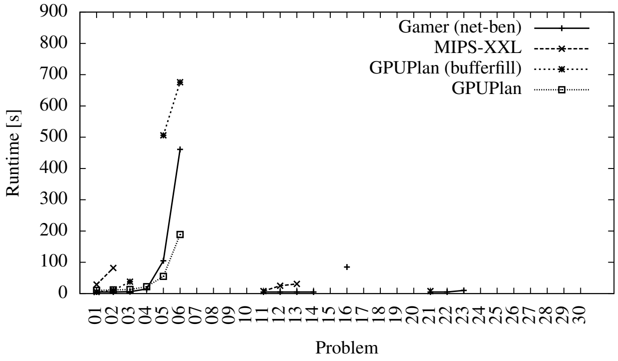


Figure 11.5: Optimal Net-Benefit Track - Transport runtimes.

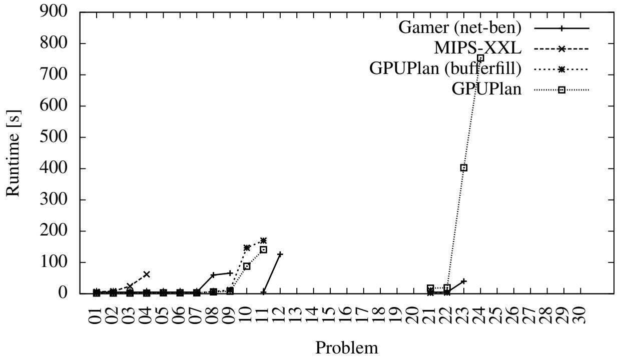


Figure 11.6: Optimal Net-Benefit Track - Woodworking runtimes.

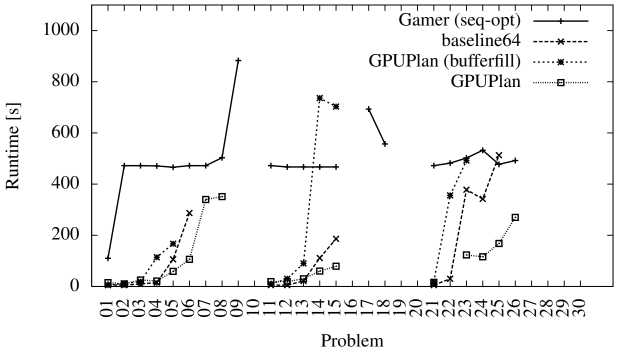


Figure 11.7: Sequential Optimal Track - Elevators runtimes.

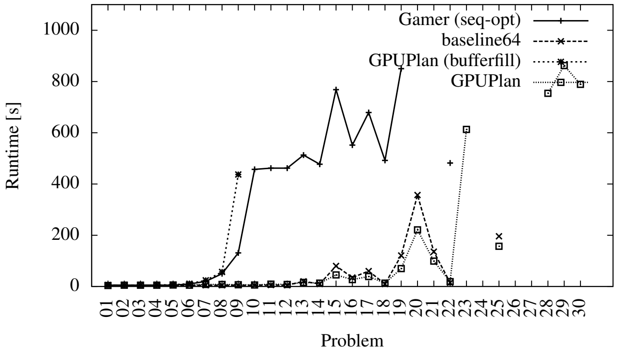


Figure 11.8: Sequential Optimal Track - Openstacks runtimes.

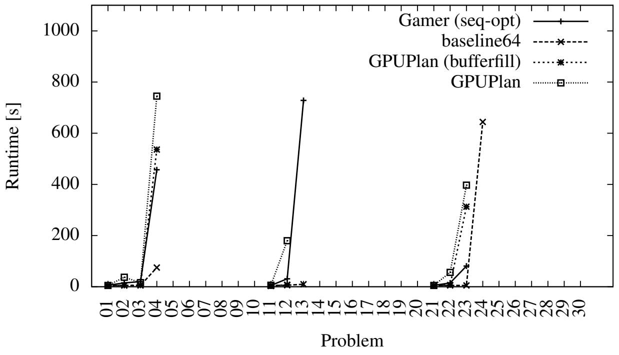


Figure 11.9: Sequential Optimal Track - Parcprinter runtimes.

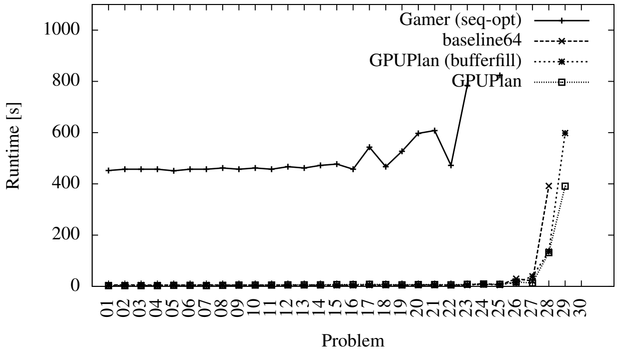


Figure 11.10: Sequential Optimal Track - Peg-Solitaire runtimes.

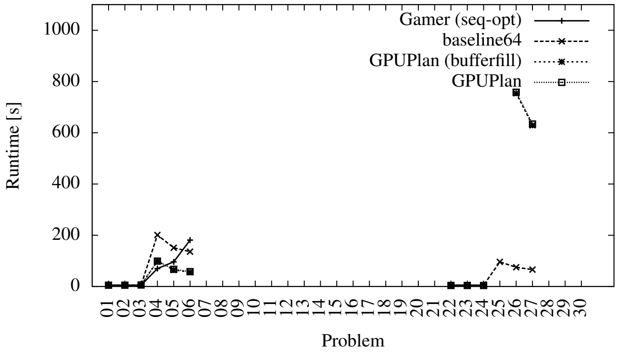


Figure 11.11: Sequential Optimal Track - Scanalyzer runtimes.

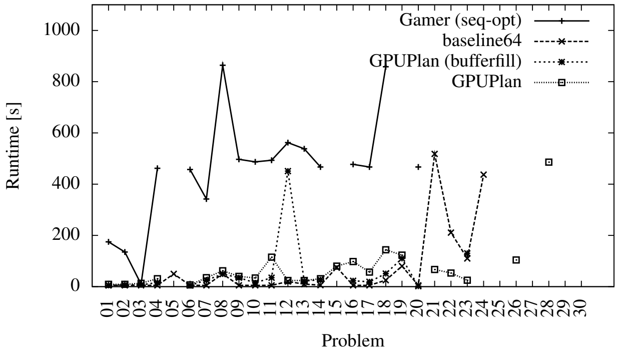


Figure 11.12: Sequential Optimal Track - Sokoban runtimes.

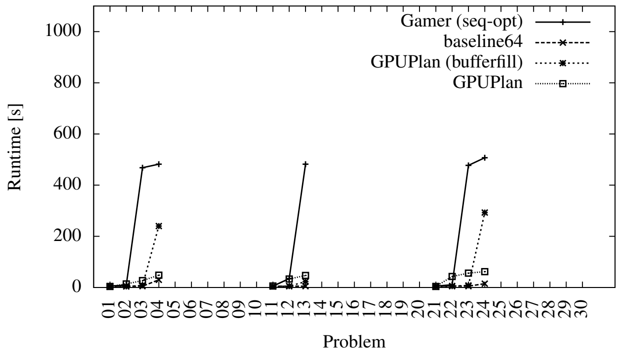


Figure 11.13: Sequential Optimal Track - Transport runtimes.

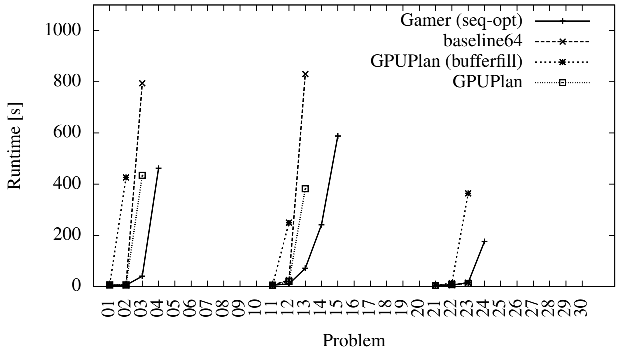


Figure 11.14: Sequential Optimal Track - Woodworking runtimes.

11.2 Summary

This chapter depicts the evaluation of the GPU enhanced planning approach. The buffer-filling algorithm and a non-filling approach are compared to state of the art planners on different domains taken from the international planning competition 2008. The plots present an efficient planner for all domains providing an increasing speedup with the increasing complexity of the problem. Solely small search spaces are inspected slower then by the rivalry due to the lag introduced by copying the state into the graphics cards memory. The plots also depicted that the efficiency of the buffer-filling algorithm highly depends on the structure of the state space.

Part IV

Game Solving

Chapter 12

Introduction to Game Solving

Experiences in Model Checking and Action Planning showed the advance of using a highly parallel computing device for expanding states and searching though large state spaces. An efficient compression e. g., a fixed size description, like a mapping of a state to an ordinary number increases the parallelism on the GPU. In the search of such a compression *Perfect Hashing* as used in Section 6.1 for LTL Model Checking showed promising results compressing states to about 1.4 bit per state in average. However, such a *perfect hash function* is created solely for the set of reachable states and this set has to be either generated or available prior to the search.

The second disadvantage of such a hash function is the condition to have it available on the GPU, limiting the remaining space for the computations of the states. To eliminate this disadvantage utilizing the computation power of a graphics card, a hash function is needed which utilizes no memory but can be computationally intensive. Ideally for the GPU the calculation of the hash value and the creation of the state representation given the hash value should be linear in time and independent from the state.

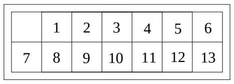
Unfortunately such compression strategies are not available nowadays for extremly variable states like in the Model Checking or Planning domain. To use a precomputed hash function distinct knowledge on the problem is required. Limiting the number of supported problems, to enable the usage of perfect hashing, results in a highly specialized, thus a highly optimized algorithm for a small number of problems, or even for a single problem e. g., enumerating the state space of a game. Here, scientists invest a large amount of resources to classify each decision point in the game and find a strategy to play the game optimally. The approach presented here was published in (Edelkamp and Sulewski, 2009) and extended from permutation games to games with indistinguishable pieces in (Edelkamp *et al.*, 2010c). The solution of Nine-Men-Morris was presented in (Edelkamp *et al.*, 2010b).

Structure of the chapter: In the next section the analyzed games are presented in detail to illustrate special aspects of them usable to generate a perfect hash function. In the remainder of this chapter available techniques for Game Solving are introduced and an example is given.

12.1 Analyzed Games

This section presents the analyzed games and sketches some initial ideas on the size of each state space. The games are classified into three groups. The first three games, the *Sliding-tile* puzzle, the *Top-Spin* puzzle and the *Pancake* problem are *one person permutation games* using a number of distinguishable pieces which have to be arranged in a special order. The following two games, namely *Peg-Solitaire* and *Frogs and Toads* are also games played by one person but the pieces are indistinguishable (*one player combinatorial games*) and the last game *Nine-Men-Morris* is a two person game with indistinguishable pieces for each player.

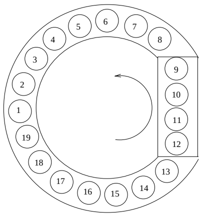
12.1.1 Sliding-Tile Puzzle



	1	2	3	4	5	6
7	8	9	10	11	12	13

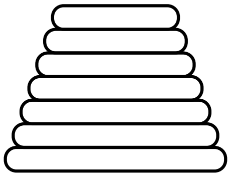
The $(n \times m)$ Sliding-Tile Puzzle (Hordern, 1986) consists of $(nm - 1)$ numbered tiles and one empty position, called the blank. In many cases, the tiles are squarely arranged, such that $m = n$. The task is to re-arrange the tiles such that a certain goal arrangement is reached.

12.1.2 Top-Spin Puzzle



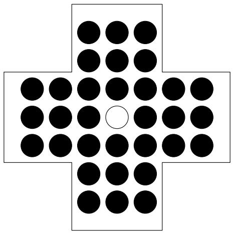
The (n, k)-Top-Spin Puzzle from Chen and Skiena (1996) has n tokens to be ordered in a ring. In one twist action of the box k consecutive tokens are reversed and in one slide action of the whole ring pieces are shifted around. There are $n!$ different possible ways to permute the tokens into the locations. However, since the puzzle is cyclic only the order of the different tokens matters and thus there are only $(n-1)!$ different states in practice. After each of the n possible actions, the permutation is normalized by cyclically shifting the array until token 1 occupies the first position in the array.

12.1.3 Pancake Problem



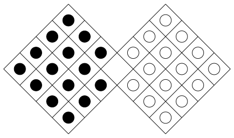
The n-Pancake Problem (Dweighter, 1975) is to determine the number of flips of the first k pancakes (with varying $k \in \{1, \ldots, n\}$) necessary to put them into ascending order. The problem has been analyzed e. g., by Gates and Papadimitriou (1979). It is known that $(5n + 5)/3$ flips always suffice, and that $15n/14$ flips are necessary. In the burned pancake variant, the pancakes are burned on one side and the additional requirement is to bring all burned sides down. It is known that $2n - 2$ flips always suffice and that $3n/2$ flips are necessary. Both problems have n possible operators. The pancake problem has $n!$ reachable states, the burned one has $n!2^n$ reachable states.

12.1.4 Peg-Solitaire



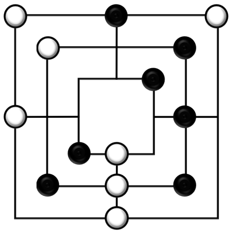
Peg-Solitaire is a single-agent problem invented in the 17th century. The game asks for the minimum number of pegs that is reachable from a given initial state. The set of pegs is iteratively reduced by jumps where the peg being jumped over is removed. Different board sizes and even structures exists in this game. Solutions for the initial state (shown in the left Figure) with one peg removed in the middle of the board are widely known (Berlekamp *et al.*, 1982). An optimal player for all possible states has been generated by Edelkamp and Kissmann (2007).

12.1.5 Frogs and Toads



The Frogs and Toads puzzle is a generalization of the Fore and Aft puzzle, which has been made popular by the American puzzle creator Sam Loyd. It is played on a part of the 7×7 board consisting of two 4×4 sub arrays at diagonally opposite corners. They overlap in the central square. One array has 15 black pieces and the other has 15 white pieces, with the center left vacant. A move is to slide or jump over another pieces of any color (without removing it). The objective is to reverse the positions of pieces in the lowest number of moves. This game was originally an English invention in the 18th century. Henry Ernest Dudeney discovered a quickest solution for two 3×3 sub arrays variant of just 46 moves (Ball, 1911).

12.1.6 Nine-Men-Morris



The game *Nine Men's Morris* has a board of three concentric squares that are connected at the mids of their sides. The 12 corner and 12 side intersections are the game positions (see Figure). Initially, each player picks 9 pieces in one color. The game divides into a set (I), a move (II) and a jump (III) phase. In all phases a player may close *mills*, i. e., align three pieces in his color horizontally or vertically. In this case, he can remove one of the opponent's pieces from the board provided that it is not contained in a mill.[1] The game is won, when the number of opponent's pieces has been reduced to 2.

By an exhaustive enumeration on a parallel architecture, the game was shown to be a draw by Gasser (1996), but his results have never been validated. He partitioned the state space in sets S_{k_1,k_2}, which contained k_1 pieces of the first player, k_2 pieces of the

[1] For the case of having two mills closed in one move, only one piece can be taken, and if the opponent only has mills, they can be destroyed.

second player and $n - (k_1 + k_2)$ empty intersections. Obviously, S_{k_1,k_2} and S_{k_2,k_1} are symmetrical, so that it is sufficient to consider S_{k_1,k_2} with $k_1 \geq k_2$ only.

12.2 Game Solving

Solving a game describes the task to classify each state of a game whether it is solvable, in single-agent games, or whether a player can still win the game, in multi-player games. The state space of a game is a tree with the initial state being the root and the leafs being the terminal states where either the game is won, not won or a draw, denoted as the *game-theoretical value* for each state.

In game theory (van den Herik *et al.*, 2002), a *zero-sum game* is a mathematical representation of a game in which the outcome for all players sums up to zero. Flipping a coin[2] is an obvious zero-sum game, since one player wins (e. g., denoted as 1 in the state) and the other player looses (denoted as 0 in a state). The sum of this game is always zero. In contrast, a *non-zero-sum* describes a game like Mikado. Here each player picks up sticks and the winner is the player having picked up more sticks.

When a forward search is not sufficient to determine the game-theoretical value, a search backward is performed. This *retrograde analysis* starts in states where the outcome is known and continues generating parents from successors. When reachable states from the initial are known (e. g., from a previous forward search) this knowledge is used to ignore not reachable states.

Figure 12.1 illustrates a part of the state space for the two-agent game *Tic-Tac-Toe* where the players put a circle or a cross on a 3 board. Winner is the player who can put 3 own tokens in a row (horizontally, vertically or diagonally). The game starts with an empty grid and the first player puts a cross on the board, resulting in a branching factor of 9. The figure shows only the branch of the cross being in the central position of the grid. Now, in the following round, the second player puts a circle anywhere on the grid, but on occupied places, resulting in a branching factor of 8. The theoretical size of the state space is $9! = 362.880$. However, the game stops when one player has reached 3 tokens in a row, removing this states results in a size of 255,168. Exploring symmetries can even shrink it down to 26,830.

Usually the solution to a game is an enumerated state space, where each state is enriched with a rule on how to proceed in the game optimally. Figure 12.2[3] shows an alternative way to present a solution for the Tic-Tac-Toe game. The first player draws a cross on the grid where the bold cross is located in the solution, and zooms in to the adequate square when the second player has drawn his circle. When both players play optimally the game ends in a draw.

12.3 Related Work

Over the past few years, researchers have shown that the scalability of state space search algorithms can be dramatically improved by using external memory, such as

[2] two players try to predict the upper side of a coin after it flips.

[3] http://xkcd.com/832/

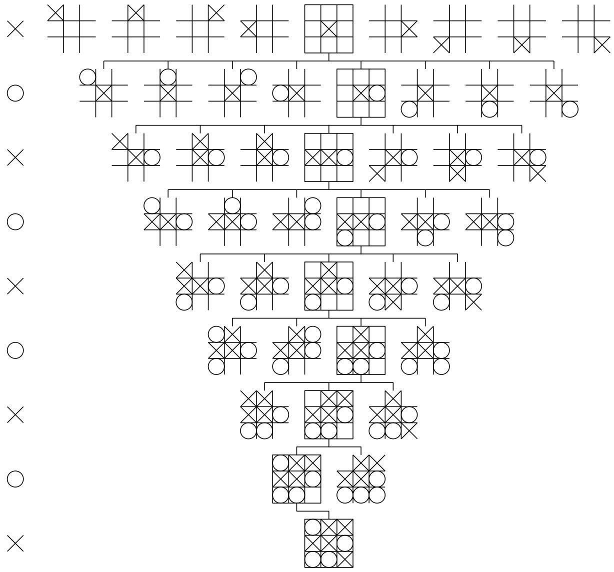


Figure 12.1: Part of the state space for the game Tic-Tac-Toe. Initially the 3 × 3 grid is empty, then each player puts a cross and a circle on the grid. The game ends with a draw. Note, there is a winning state for the circle player in the last but one row. Boards in a box denote the chosen path.

disks, to store generated states for duplicate detection. However, this requires very different search strategies to overcome the six orders-of-magnitude difference in random-access speed between RAM and disk.

Munagala and Ranade (1999) suggested an I/O-efficient breadth-first search (BFS) algorithm for explicit graphs with adjacencies that are stored on disk, while Korf (2003b) applied a related disk-based frontier BFS algorithm to explore implicit state spaces that are generated by the repeated application of moves. The duplicate detection schemes in these frontier search algorithms can either be delayed (Korf, 2008a) or structured (Zhou and Hansen, 2004).

As RAM remains a scarce resource, external-memory two-bit BFS by Korf (2008b) integrates the compression method by Cooperman and Finkelstein (1992) into an I/O-efficient algorithm. The approach for solving large-scale problems relies on perfect hash functions. It applies a space-efficient bit vector representation of the state space with two bits per state, and allows states to be revisited. As a result, Kunkle and Cooperman (2007) could prove a bound for solving Rubik's Cube of 26 moves.

The above approaches scale especially well on multiple disks (e.g., combined in a soft- or hardware RAID). Here, the computational bottleneck is shifted back to the internal computing time instead of being dominated by transferring the data due to disk latencies. This has let to another change of the focus in AI research: a rising number of parallel search variants has been studied, e.g., by Korf (2008a); Zhou and Hansen (2007); Burns *et al.* (2009a), as well as by Jabbar and Edelkamp (2006).

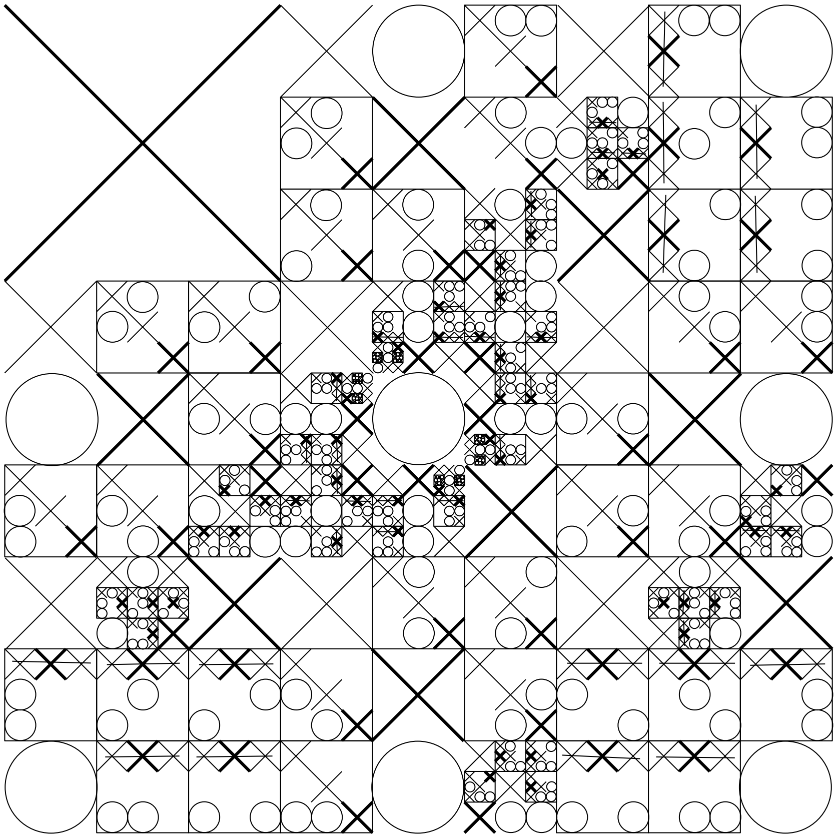


Figure 12.2: Solution to the Tic-Tac-Toe game when cross starts wit the upper right corner. Circle can not win and there are only 4 draw endings. As the first player, draw your cross where the bold cross is. Zoom in to the square where the second player has drawn his circle.

12.4 Summary

This chapter, the starting chapter for the Game Solving part introduces chosen games and groups them according to the number of players and the discriminability of the used pieces. The second section introduces the aspects of Game Solving and introduces terms common in this discipline. The chapter closes with a sketch on related work in parallel game solving.

Chapter 13

Perfect Hashing in Games

Having introduced the games and solution generation strategies this chapter presents specific perfect hashing methods. According to the definition in Section 1.4.1 a minimal perfect hash function is a one-to-one mapping from the state space S to the set $\{0,\ldots,|S|-1\}$. For many AI search problem domains perfect hash functions and their inverses are available prior to the search. By analysing the state space a partitioning can be extracted which enables perfect hashing due to a beneficial segmentation. One example of such a partitioning in the real world is the classification of the world according to the *Global Positioning System* (GPS) coordinates. Here two hash functions exist (h_x, h_y) with defined starting points and distances, the concatenation of both enables to identify an unique position on the globe. Naturally such a partitioning can not be defined easily for a state space but a deeper analysis of the game can reveal *anchor points* for the desired function.

Structure of the chapter: Having proposed properties for general state spaces in games, three perfect hashing methods will be presented optimized for each aforementioned groups of games. The chapter closes with a proposal to use perfect hashing to solve games.

13.1 Properties of State Spaces in Games

For state spaces in games properties can be defined to partition the state space and generate perfect hash functions by applying the fact that orthogonal hashing implies perfect hashing. The first such property shown here is the *Move-Alternation* property dividing the state space in two distinct partitions.

Definition 31 (Move-Alternation) *Property $p : S \rightarrow \{0,1\}$ is* move-alternating*, if it toggles for all applied actions.*

In other words, for all successors s' of s, $p(s') = 1 - p(s)$. As a result, $p(s)$ is the same for all states s in one BFS-Layer, so that states s' in the next BFS-Layer can be separated from the ones in the current one, exploiting $p(s') = x \neq y = p(s)$. A stronger criterion is the following one.

Definition 32 (Layer-Selection) *A property $p : S \rightarrow \mathbb{N}$ is* layer-selecting*, if it determines the BFS-Layer for a state, in other words $p(s) =$* BFS-Layer(s).

An example is the number of unoccupied holes in the *Peg-Solitaire* game. It starts with one and increases by one with each move. In some cases perfect hash functions can be partitioned along the properties (Korf and Schultze, 2005).

Definition 33 (BFS-Partitioning) *A perfect hash function h is* alternation partitioning*, if there is a move-alternation property p that is orthogonal to h. A perfect hash function h is* layer partitioning*, if there is a layer-selection property p that is orthogonal to h.*

For a given perfect hash function h for the full state space this leads to further compression, and in some cases memory advances when applying frontier search, depending on the *locality* of the search space (Zhou and Hansen, 2006).

If p is a move-alteration property, S can be partitioned into parts $S_0 = \{s \mid p(s) = 0\}$ and $S_1 = \{s \mid p(s) = 1\}$ with $S_0 \cup S_1 = S$ and $S_0 \cap S_1 = \emptyset$, such that $h(s_0) < h(s_1)$ for $(s_0, s_1) \in S_1 \times S_2$. This defines two bit-vector compression functions $h_0(s) = h(s)$ and $h_1(s) = h(s) - |\{h(s) \mid s \in S_0\}|$ that can be used in odd and even layer of the search. Similarly, the observation can be extended to hash functions $h_1, h_2, h_3, \ldots$, if p is layer partitioning.

For many domain-independent problem domains, perfect hash functions (and their inverses) can be derived.

13.2 Ranking and Unranking in Permutation Games

For the design of rank and unrank functions for permutation games parity is a crucial concept.

Definition 34 (Parity) *The* parity *of the permutation π of length N is defined as the parity of the number of inversions in π, where inversions are all pairs (i, j) with $0 \leq i < j < n$ and $\pi_i > \pi_j$.*

Definition 35 (Parity Preservation) *A permutation problem is* parity preserving*, if all moves preserve the parity of the permutation.*

Parity Preservation allows separating solvable from insolvable states in several permutation games. Examples are the sliding-tile and the (n, k) Top-Spin puzzles (with even value of k and odd value of n). If the parity is preserved, the state space can be compressed.

In all permutation games the time for generating a successor is dominated by the time for ranking and unranking. For ranking and unranking permutations, efficient time and space algorithms have been designed (Bonet, 2008). The design of a minimal perfect hash function for the sliding-tile puzzle, can be observed that in a lexicographic ranking every two adjacent permutations π_{2i} and π_{2i+1} have a different solvability status.

Definition 36 (Lexicographic Rank, Inverted Index) *The lexicographic rank of permutation π (of size N) is defined as $\sum_{i=0}^{N-1} d_i \cdot (N-1-i)!$ where the vector coefficients d_i are called the* inverted indexes.

The coefficients d_i are uniquely determined. The parity of a permutation is known to match $(\sum_{i=0}^{N-1} d_i)$ *mod* 2.

In order to hash a sliding-tile puzzle state to $\{0,\ldots,(nm)!/2-1\}$, the lexicographic rank can be computed and divided by two. Unranking is slightly more complex, as it has to determine, which of the two permutations π_{2i} and π_{2i+1} of the puzzle with index i is reachable.

Korf and Schultze (2005) use two lookup tables to compute lexicographic ranks with a space requirement of $O(2^N \log N)$ bits. Bonet (2008) discusses time-space trade-offs and provides an uniform algorithm that takes $O(N \log N)$ time and $O(N)$ space. An evaluation revealed that existing ranking and unranking algorithms wrt. the lexicographic ordering are rather slow. Hence, the more efficient ordering of Myrvold and Ruskey (2001) was studied in more detail, and show that the parity of a permuation can be derived on-the-fly.[1] For faster execution (on the graphics card) recursion is avoided (see Algorithm 13.1).

Algorithm 13.1: *unrank*(r) with parity derived on-the-fly

```
Input: $r$ rank of a state
Output: $(parity, \pi)$ parity and reconstructed permutation
$\pi \leftarrow id$ ;                      {store ordered sequence in $\pi$}
$parity \leftarrow false$ ;                {initialize $parity$}
while $N > 0$ do                           {until all elements processed}
    $i \leftarrow N-1$ ;                   {position of first element}
    $j \leftarrow r$ mod $N$ ;             {position of second element}
    if $i \neq j$ then                     {count only different positions}
        $parity \leftarrow \neg parity$ ;  {negate $parity$}
        $swap(\pi_i, \pi_j)$ ;             {swap elements at the positions}
        $r \leftarrow r$ div $N$ ;         {remove swapped position from the rank}
    $N \leftarrow N-1$ ;                   {proceed with the next position}
return $(parity, \pi)$ ;                   {return parity and permutation}
```

Theorem 4 (Parity in Myrvold & Ruskey's Unrank function) *The parity of a permutation given a rank in Myrvold & Ruskey's ordering can be computed on-the fly in the unrank function depicted in Algorithm 13.1.*

Proof. In the *unrank* function swapping two elements u and v at position i and j, resp., with $i \neq j$ $2(j-i-1)+1$ transpositions exist (u and v are the elements to be swapped,

[1] This work always refers to Myrvold and Ruskey's rank1 and unrank1 functions.

x is a wild card for any intermediate elements): $uxx\ldots xxv \to xux\ldots xxv \to \ldots \to xx\ldots xxuv \to xx\ldots xxvu \to \ldots \to vxx\ldots xxu$. As $2(j-i-1)+1$ *mod* $2=1$, each transposition either increases or decreases the parity of the number of inversions, so that the parity toggles for each iteration. The only exception is if $i=j$, where no change occurs. Hence, the parity of the permutation can be determined on-the-fly in our algorithm.

□

Theorem 5 (Folding Myrvold & Ruskey) *Let $\pi(r)$ denote the permutation returned by Myrvold & Ruskey's* unrank *function given index r. Then $\pi(r)$ matches $\pi(r+N!/2)$ except for swapping π_0 and π_1.*

Proof. The last call to *swap*$(\pi_{N-1}, \pi_{r\ mod\ N})$ in Myrvold and Ruskey's *unrank* function is *swap*$(\pi_0, \pi_{r\ mod\ 1})$, which resolves to either *swap*(π_1, π_1) or *swap*(π_1, π_0). Only the latter one induces a change.

If $r_1, \ldots, r_{N-1}$ denote the indexes of r *mod* N in the iterations $1, \ldots, N-1$ of Myrvold and Ruskey's *unrank* function, then $r_{N-1} = \lfloor \ldots \lfloor r/(N-1) \rfloor \ldots /2 \rfloor$, which resolves to 1 for $r \geq N!/2$ and 0 for $r < N!/2$. □

13.2.1 Reducing State Space in Permutation Games

Sliding-Tile Puzzle

Swapping two tiles toggles the permutation parity and in turn the solvability status of the game. Thus, only half the states are reachable.

There is one subtle problem with the blank. Simply taking the parity of the entire board does not suffice, as swapping a tile with the blank is a move, which does not change it. A solution is to partition the state space wrt. the position of the blank, since for exploring the $(n \times m)$ puzzle it is equivalent to enumerate all $(nm-1)!/2$ orderings together with the nm positions of the blank. If $B_0, \ldots, B_{nm-1}$ denote the set of "blank-projected" partitions, then each set B_i contains $(nm-1)!/2$ states.

Top-Spin Puzzle

Depending on the value k and an odd value of n, a twist will always change the parity or not. Given an even value of k (the default), only a twist on token 1 may change the parity.

Theorem 6 (Parity Anomaly in Top-Spin) *For an even value of k and odd value of $n > k+1$, the (normalized) (n,k) Top-Spin Puzzle has $(n-1)!/2$ reachable states.*

Proof. To ease notation, w.l.o.g., the proof is done for $k=4$. Let $n=2m+1$ and $(x_0, x_1, \ldots, x_{2m})$ be the normalized state vector. Thus, due to normalization, $x_0=0$. First of all, given that 0 is not counted, only three elements change their position and lead to 3 transpositions. For $(0, x_1, x_2, x_3, \ldots, x_{2m})$ four critical successor states exist:

- $(x_3, x_2, x_1, 0, x_4 \ldots . x_{2m})$,

- $(x_2, x_1, 0, x_{2m}, x_3, ..., x_{2m-1})$,
- $(x_1, 0, x_{2m}, x_{2m-1}, x_2, ..., x_{2m-2})$, and
- $(0, x_{2m}, x_{2m-1}, x_{2m-2}, x_1, ..., x_{2m-3})$.

In all cases, normalization has to move 3 elements either the ones with low index to the end of the array to postprocess the twist, or the ones with large indexes to the start of the array to preprocess the operation. The number of transpositions for one such move is $2m - 1$. In total $3(2m - 1) + 3$ transpositions exist. As each transposition changes the parity and the total of $6m$ transpositions is even, all critical cases have even priority. □

As the parity is odd and even for a move in the (normalized) (n, k) Top-Spin Puzzle for an odd value of $n > k + 1$, the entire set consists of $(n - 1)!$ reachable states.

Pancake Problem

For an even value of $\lceil (k - 1)/2 \rceil$, $k > 1$ the parity changes, while for an odd one, the parity remains the same.

13.3 Binomial Coefficient for Single Player Games

For states consisting of a fixed number of Boolean variables, it suffices to store only the variables that are assigned `true`, in order to identify each state. Traversing the search graph and generating successors flips the status of individual state variables depending on the successor generating function.

If the order of the variables is fixed and the number of satisfied bits are given, their position can be identified using a *binomial coefficient*. A binomial coefficient $\binom{n}{k}$ is the number of possible k-sets in a set of n elements. Algorithm 13.2 describes how to assert an unique rank to a given state. Since the number of k-sets in a n-set is known, an ordering can be imposed on these k-sets. This ordering is given by the position of the variables that are satisfied. The algorithm starts with a rank $r = 0$ and uses the variable t to count the number of satisfied variables. For each unsatisfied variable r is increased by the binomial coefficient given by the position of this entry and the number of the remaining satisfied variables.

The according unrank function is displayed in Algorithm 13.3. For proving the correctness surjectivity and injectivity is shown.

Theorem 7 (Surjectivity) *For each $s \in S$ with t satisfied variables, the maximal rank is bounded by $r(s) \leq \binom{n}{n-t} - 1$.*

Proof. Value r is increased only if no satisfied variable is found. For all $n, t \in \mathbb{N}$ with $n \geq t$, $\binom{n}{t} \geq \binom{n-1}{t}$. We get the maximal rank when the first $n - t$ variables are not satisfied, resulting in $r = \binom{n-1}{t-1} + \ldots + \binom{n-t}{t-1}$. Using the known characteristic of binomial coefficients $\sum_{i=0}^{m} \binom{n+i}{n} = \binom{n}{n} + \ldots + \binom{n+m}{n} = \binom{n+m+1}{n+1}$

$\sum_{i=0}^{t} \binom{(}{n} - t) - 1 + i(n-t) - 1 = \binom{n}{n-t}$, and, by applying Algorithm 13.2, $\binom{n-t}{(n-t)-1} + \ldots + \binom{n-1}{(n-t)-1} = \binom{n}{n-t} - \binom{(n-t)-1}{(n-t)-1} = \binom{n}{n-t} - 1$. □

Theorem 8 (Injectivity) *Given a number of satisfied variables t binomial ranking induces a collision free hash function, such that for all $s, s' \in S_t$, $s \neq s'$ implies $r(s) \neq r(s')$.*

Proof. Assume the contrary. Then there exists $s, s' \in S_t$ with $s \neq s'$ and $r(s) = r(s')$. Since s and s' are different, an entry at a minimal position i exists with $s_i \neq s'_i$. If w.l.o.g. s_i is not satisfied, equation $r(s_{0..i}) = r(s'_{0..i}) + \binom{n-i}{t_i-1}$ applies, where t_i is the number of satisfied elements left, but the maximal increase for $r(s')$ is, due to the same arguments as in Theorem 7, $\binom{(n-j)-1}{t_i-2} + \ldots + \binom{t_i-1}{t_i-2} = \binom{n-j}{t_i-1} - 1$. □

Algorithm 13.2: Binomial-Rank

```
Input: $s$ state to rank
Output: $r$ rank
$i \leftarrow 0; r \leftarrow 0$ ;                          {reset position counter and rank}
$t \leftarrow$ number of true values in $s$ ;              {variable for backwards counting}
while $t > 0$ do
    $i \leftarrow i + 1$ ;                                   {increase position counter}
    if $s_i = 1$ then                                        {do not count true bits}
        $t \leftarrow t - 1$ ;
    else
        $r \leftarrow r + \binom{n-i}{t-1}$ ;
        {increase r by the binomial of left positions over left true bits}
return $r$ ;                                                 {return rank}
```

13.4 Multinomial Coefficient for Multi Player Games

Multinomial coefficients can be used to compress state vectors with a fixed but permuted value assignment, e. g., board game state (sub)sets where the number of pieces for each player does not change. For p players in a game on n positions k_i with $1 \leq i \leq p$ is used to denote the number of game pieces owned by player i, and k_{p+1} for the remaining empty positions.

Definition 37 *For n, k_1, k_2, …, $k_m \in \mathbb{N}$ with $n = k_1 + k_2 + \ldots + k_m$ the multinomial coefficient is defined as*

$$\binom{n}{k_1, k_2, \ldots, k_m} \leftarrow \frac{n!}{k_1! \cdot k_2! \cdot \ldots \cdot k_m!}.$$

Algorithm 13.3: Binomial-Unrank

```
Input: $r$ rank, $t$ number of bits to enable
Output: $s$ generated state
$i \leftarrow 0$ ;                                      {reset position counter}
while $t > 0$ do                                        {until all true bits generated}
    if $r < \binom{n-i-1}{t-1}$ then
        $s_i \leftarrow true$ ;                         {enable bit}
        $t \leftarrow t - 1$ ;                          {decrease number bits to enable}
    else
        $s_i \leftarrow false$ ;                        {disable bit}
        $r \leftarrow r - \binom{n-i-1}{t-1}$ ;         {decrease rank due to a set false bit}
    $i \leftarrow i + 1$ ;                              {proceed with nest position}
while $i < |s|$ do
    $s_i \leftarrow false$ ;                            {disable remaining bits}
    $i \leftarrow i + 1$;
return $s$ ;                                            {return state}
```

Since $\sum_0^{p+1} k_i = n$ value k_{p+1} can be deduced given $k_1, k_2, \ldots, k_p$. This section presents present multinomial hashing for $p = 2$ but the extension to three and more players is intuitive.

In the remainder of this work $\binom{n}{k_1,k_2}$ will be used for $\binom{n}{k_1,k_2,k_3}$ with $k_3 = n - (k_1 + k_2)$ and distinguish pieces by enumerating their *colors* with 1, 2, and 0 (empty).

Let S_{k_1,k_2} be the set of all possible boards with k_1 pieces of color 1 and k_2 pieces in color 2. The computation of the rank for states in S_{k_1,k_2} is provided in Algorithm 13.4. The intuition is that the algorithm *Multinomial-Rank* defines h_{k_1,k_2} via counting with multinomial coefficient. For each position i it checks, if a 2 (line 3), a 1 (line 5) or a 0 (line 8) is present in the state vector.

- If a 2 is found, the value in variable l_{twos} is decremented by 1, while r remains unchanged.
- In case of a 1, the according multinomial coefficient describes the number of assignments, that have been visited and that contain a 2 at the current position. This is done only if there are still remaining 2s ($l_{twos} > 0$). Since a 1 was seen variable l_{ones} is decremented by one.
- If a 0 is processed, all visited 2s are skipped as long as $l_{twos} > 0$ and all 1s up to the current position if $l_{ones} > 0$.

Theorem 9 *The hash function defined in Algorithm 13.4 is bijective.*

Proof Let $h_{k_1,k_2} : S_{k_1,k_2} \longmapsto \mathbb{N}$ be the hash function defined by Algorithm 13.4. We show: 1) for all $s \in S_{k_1,k_2}$ we have $0 \leq h_{k_1,k_2}(s) \leq \binom{n}{k_1,k_2} - 1$; and 2) for all $s, s' \in S_{k_1,k_2}$: $s \neq s'$ implies $h_{k_1,k_2}(s) \neq h_{k_1,k_2}(s')$.

Algorithm 13.4: Multinomial-Rank

```
Input: Game state vector: $s_{0,\ldots,n-1}$, number of pieces in color 1: $l_{ones}$, number
       of pieces in color 2: $l_{twos}$
Output: $r$ rank
$i \leftarrow 0; r \leftarrow 0$ ;                              {reset position counter and rank}
while $i < n$ do                                                {for each position}
    if $s_i = 2$ then
        $l_{twos} \leftarrow l_{twos} - 1$ ;            {piece of color 2 just decrease counter}
    else if $s_i = 1$ then
        if $l_{twos} > 0$ then
            $r \leftarrow r + \binom{n-i-1}{l_{ones}, l_{twos}-1}$ ;
            {piece of color 1 increase rank if remaining pieces of color 2 exist}
        $l_{ones} \leftarrow l_{ones} - 1$ ;
    if $l_{twos} > 0$ then
        $r \leftarrow r + \binom{n-i-1}{l_{ones}, l_{twos}-1}$ ;
                    {$s_i = 0$ increase rank if remaining pieces of color 2 exist}
    if $l_{ones} > 0$ then
        $r \leftarrow r + \binom{n-i-1}{l_{ones}-1, l_{twos}}$ ;
                    {$s_i = 0$ increase rank if remaining pieces of color 1 exist}
    $i \leftarrow i + 1$ ;                                      {proceed with next position}
return $r$ ;                                                    {return rank}
```

1) As r is initialized to 0 and increases monotonically, we only show the upper bound. The values that are added to r are $value_1 = \binom{n-i-1}{l_{twos}, l_{ones-1}}$ and $value_2 = \binom{n-i-1}{l_{twos}-1, l_{ones}}$. These values depend on the position $(i+1)$ of the currently considered state vector entry and on the number of non-processed pieces of color 1 (l_{ones}) and color 2 (l_{twos}). We additionally observe that the number of non-processed pieces referred to in the bottom line of the expressions decreases monotonically.

Exploiting that for all $n \in \mathbb{N}^+$, and all $k_1, k_2, k_3 \in \mathbb{N}$ with $n = k_1 + k_2 + k_3$ we have:

$$\binom{n}{k_1, k_2, k_3} \geq \binom{n-1}{k_1, k_2, k_3}$$

and for all $n, k_1 \in \mathbb{N}^+$, and all $k_2, k_3 \in \mathbb{N}$ with $n = k_1 + k_2 + k_3$:

$$\binom{n}{k_1, k_2, k_3} \geq \binom{n-1}{k_1 - 1, k_2, k_3 + 1},$$

$value_1$ and $value_2$ are maximized, if the first position of the state vector entry is maximized, followed by the second and so forth. Hence r is maximal, if at the first k_3 positions we have only 0s, while in the following k_1 positions we have only 1s and the remaining k_2 positions contain 2s.

As for such maximal r the first k_3 positions contain only 0s, the according values l_{ones} and l_{twos} in the corresponding multinomial coefficient are constant. These positions thus add the following offset $\Delta_{0,max}$ to r ($k_1 = l_{ones}$ and $k_2 = l_{twos}$):

$$\sum_{i=1}^{k_3} \left(\binom{n-i}{k_1,k_2-1,k_3+1-i} + \binom{n-i}{k_1-1,k_2,k_3+1-i} \right)$$

At the following k_1 positions for such maximal r all 1s are scanned, while the value k_2 remains constant at l_{twos}. Value l_{ones} matches k_1 initially and is decremented by 1 for each progress in i. Obviously, 0s are no longer present, such that the offset $\Delta_{1,max}$ equals

$$\sum_{i=1}^{k_1} \binom{n-k_3-i}{k_1+1-i,k_2-1,0}$$

As the multinomial coefficient can be expressed as a product of binomial coefficients.

$$\binom{n}{k_1,k_2,\ldots,k_r} = \binom{k_1+k_2}{k_2} \cdot \ldots \cdot \binom{k_1+k_2+\ldots+k_r}{k_r}$$

we rewrite the summands for $\Delta_{0,max}$ to

$$\sum_{i=1}^{k_3} \left(\binom{k_1+k_2-1}{k_2-1} \binom{n-i}{k_3+1-i} \right)$$

and

$$\sum_{i=1}^{k_3} \left(\binom{k_1+k_2-1}{k_2} \binom{n-i}{k_3+1-i} \right)$$

For binomial coefficients we have

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

and

$$\sum_{i=1}^{k_3} \binom{n-i}{k_3+1-i} = \sum_{i=1}^{k_3} \binom{n-k_3-1+i}{n-k_3-1}$$

This implies that $\Delta_{0,max}$ is equal to

$$\begin{aligned}
&\sum_{i=1}^{k_3} \left(\binom{n-i}{k_3+1-i} \cdot \left(\binom{k_1+k_2-1}{k_2-1} + \binom{k_1+k_2-1}{k_2} \right) \right) \\
&= \sum_{i=1}^{k_3} \left(\binom{n-i}{k_3+1-i} \binom{k_1+k_2}{k_2} \right) \\
&= \binom{k_1+k_2}{k_2} \cdot \left(\binom{n}{n-k_3} - 1 \right)
\end{aligned}$$

and $\Delta_{1,max}$ is equal to

$$\begin{aligned}
&\sum_{i=1}^{k_1} \left(\binom{k_1-i+1}{k_1-i+1} \binom{k_1+k_2-i}{k_2-1} \binom{k_1+k_2-i}{0} \right) \\
&= \sum_{i=1}^{k_1} \binom{k_1+k_2-i}{k_2-1} \\
&= \binom{k_1+k_2}{k_2} - 1.
\end{aligned}$$

Algorithm 13.5: Multinomial-Unrank

```
Input: $r$ rank , number of pieces color 1: $l_{ones}$, number of pieces color 2: $l_{twos}$
Output: $s_{0\dots n-1}$ Game state vector
$i \leftarrow 0$ ;                                              {reset position counter}
while $i < n$ do
    if $l_{twos} > 0$ then
        $value_2 \leftarrow \binom{n-i-1}{l_{ones}, l_{twos}-1}$ ; {2s left, store binomial coefficient for $s_i = 2$}
    else
        $value_2 \leftarrow 0$ ;                                {all 2s set}
    if $l_{ones} > 0$ then
        $value_1 \leftarrow \binom{n-i-1}{l_{ones}-1, l_{twos}}$ ; {1s left, store binomial coefficient for $s_i = 1$}
    else
        $value_1 \leftarrow 0$ ;                                {all 1s set}
    if $r < value_2$ then
        $s_i \leftarrow 2$ ;                                    {set when $0 \leq r < value_2$}
        $l_{twos} \leftarrow l_{twos} - 1$ ;
    else if $r < value_1 + value_2$ then
        $s_i \leftarrow 1$ ;                          {set when $value_2 \leq r < value_1 + value_2$}
        $r \leftarrow r - value_2$ ;
        $l_{ones} \leftarrow l_{ones} - 1$ ;
    else
        $s_i \leftarrow 0$ ;                          {set when $value_1 + value_2 < r$}
        $r \leftarrow r - (value_1 + value_2)$ ;
    $i \leftarrow i + 1$ ;
return $state$ ;
```

Hence, the maximal possible value for r is

$$\begin{aligned} r_{max} &= \Delta_{0,max} + \Delta_{1,max} \\ &= \binom{k_1+k_2}{k_2} \cdot \left(\binom{n}{n-k_3} - 1\right) + \binom{k_1+k_2}{k_2} - 1 \\ &= \binom{k_1}{k_1}\binom{k_1+k_2}{k_2}\binom{n}{k_3} - 1 \\ &= \binom{n}{k_1, k_2, k_3} - 1. \end{aligned}$$

2. Consider two states $s_1, s_2 \in S_{k_1,k_2}$ and the smallest possible index in which the two states differ, i. e.,

$$i' := \min\,\{i \mid 0 \leq i \leq (n-1) \ \wedge \ state_{s_1}[i] \neq state_{s_2}[i]\}$$

The values of r computed up to i' are the same. Let ${k'}_1 \leq k_1$ and ${k'}_2 \leq k_2$ be the remaining pieces of the respective color. We have $r_{s_1,i'} = r_{s_2,i'}$, where $r_{s,i'}$ denotes the value r computed for state s before evaluating position i'. At position i' we have the following three cases.

In the first case, $state_{s_1}[i'] = 0$ and $state_{s_2}[i'] = 1$. The difference of the r values is

$$r_{s_1,i'+1} = r_{s_2,i'+1} + \binom{n-i'-1}{k'_1 - 1, k'_2}$$

Following the above derivations, we know that r_{s_2} increases by at most $\binom{n-i'-1}{k'_1-1,k'_2} - 1$ in the $(n - i' - 1)$ remaining positions with $(k'_1 - 1)$ and k'_2 pieces of the according color, such that $r_{s_1,j} \neq r_{s_2,j}$ for $j > i'$.

In the second case, we have $state_{s_1}[i'] = 0$ and $state_{s_2}[i'] = 2$. This implies

$$r_{s_1,i'+1} = r_{s_2,i'+1} + \binom{n-i'-1}{k'_1 - 1, k'_2} + \binom{n-i'-1}{k'_1, k'_2 - 1}$$

Value r_{s_2} increases by at most $\binom{n-i'-1}{k'_1,k'_2-1} - 1$ on the remaining $(n - i' - 1)$ positions with k'_1 and $(k'_2 - 1)$ pieces of the according color. We again have $r_{s_1,j} \neq r_{s_2,j}$ for $j > i'$.

The remaining case is $state_{s_1}[i'] = 1$ and $state_{s_2}[i'] = 2$ with

$$r_{s_1,i'+1} = r_{s_2,i'+1} + \binom{n-i'-1}{k'_1, k'_2 - 1},$$

where the argumentation of the second case applies. □

Algorithm 13.5 is the inverse of Algorithm 13.4 and used to compute $h^{-1}_{k_1,k_2}$ in form of assignments to a state vector. As the Unrank procedure subtracts the multinomial coefficients that match the ones that have been added in *Rank*, the inverse $h^{-1}_{k_1,k_2}$ is computed correctly.

13.5 Summary

Having defined a number of properties common in the state spaces of games, the chapter continued with three propositions according to the groups of introduces games. Ranking and unranking can be used as a perfect hash function in permutation games, while binominal and multinomial coefficients are suitable for games with indistinguishable pieces. This part continues with a proposal of algorithms to enumerate state spaces in games.

Chapter 14

GPU Enhanced Game Solving using Perfect Hashing

Based on computing reversible minimal perfect hash functions on the GPU, one-bit reachability and one-bit BFS algorithms are proposed. Specific perfect hash functions are studied. In solving Nine-Men-Morris a speed-up factor of over 12 is obtained. Specialized hashing for ranking and unranking states on the GPU and a parallel retrograde analysis on the GPU is applied. Unfortunately, the AI exploration approaches hardly carries over to Model Checking, as general designs of invertible hash functions – as available for particular games – are yet unknown.

Structure of the chapter: This chapter introduces techniques to enumerate all states in a state space by traversing it in a BFS manner. A BFS approach is presented utilizing only two bits per state. This approach is modified to use even one-bit when only a reachability analysis is performed. In special cases when a move-alternation property exist even an one-bit Breadth-First search can be performed. Having presented the algorithms, they are ported to the GPU and the solution of the game Nine-Men-Morris is provided, also ported to the GPU.

14.1 State Space Algorithms utilizing Perfect Hashing

Section 1.4.1 already mentioned an approach to compress the $Closed$ list to only one bit in state space searching, provided a perfect hash function is available. Given the perfect hash function is also revertible (Definition 22) $Open$ can also be compressed since state s can be reconstructed using $h(s)$. This section will propose several BFS algorithms optimized for a reversible perfect hash function and reducing the space consumption of ($Open$ + $Closed$) to one bit per state.

14.1.1 Two-Bit Breadth-First search

In the domain of Caley graphs, Cooperman and Finkelstein (1992) show that, given a perfect and invertible hash function, two bits per state are sufficient to conduct a complete breadth-first exploration of the search space. The running time of their approach (shown in Algorithm 14.1) is determined by the size of the search space times the maximum breadth-first layer, times the efforts to generate the children. Each node is expanded at most once. The algorithm uses two bit encoding numbers from 0 to 3, with 3 denoting an unvisited state, and 0,1,2 denoting the current depth value modulo 3. The main effect is that this allows to distinguish newly generated states and visited states from the current layer.

For non-minimal perfect hash functions, determining all reachable states is important to distinguish the good from the bad ones. This includes filtering of terminal states in two player games like Tic-Tac-Toe. Here 5,478 states are reachable. A simple hash function maps Tic-Tac-Toe positions to $|\{O, X, -\}|^9 =$ 19,683. In this case, the efficiency is $\lceil$ 19,683/5,478 $\rceil = 4$ so that this implicit representation is fortunate compared to explicit representations which need more bits per state.

A complete BFS traversal of the search space is very important for the construction of pattern databases. Korf (2008b) has applied the algorithm to generate the state spaces for hard instances of the Pancake problem I/O efficiently.

Algorithm 14.1: Two-Bit-Breadth-First search (*init*)

```
Input: $\hat{s} \in \mathcal{S}$: initial state, $N! - 1$ number of 2 bit entries in $Open$
forall the $i \leftarrow 0, \ldots, N! - 1$ do                {initialize $Open$ }
    $Open[i] \leftarrow 3$ ;
$Open[rank(\hat{s})] \leftarrow level \leftarrow 0$ ;                {mark initial as open}
while $Open$ has changed do                {states marked as open}
    $level \leftarrow level + 1$ ;                {count the number of levels}
    forall the $i \leftarrow 0, \ldots, N! - 1$ do                {scan for states to expand}
        if $Open[i] =$ (level $- 1$) mod 3 then
        {if this is a state from the previous level}
            $s \leftarrow unrank(i)$ ;                {reconstruct state $s$ }
            expand $s \rightarrow s_1 \ldots s_\nu$ ;                {generate successors}
            for $s_j$ $(\forall j : 1 \le j \le \nu)$ do                {check each successor}
                if $Open$[rank$(s_j)] = 3$ then                {this state is new}
                    $Open[rank(s_j)] \leftarrow level$ mod 3 ;
                    {add it to the present level}
```

Two-bit Breadth-First search indicates the use of Bitstate tables for compressed pattern databases. If the mod-3 value of the BFS-level is stored, its absolute value is determined by backward construction of its path . One shortest path predecessor with mod-3 value of BFS-level k appears in level $k - 1$ *mod* 3. Reaching the initial state, the pattern database lookup-values can then be determined incrementally.

14.1.2 One-Bit Reachability

The simplification in Algorithm 14.2 allows to generate the entire state space using one bit per state.

Algorithm 14.2: One-Bit reachability (*init*)

```
Input: $\hat{s} \in \mathcal{S}$: initial state, $N! - 1$ number of 2 bit entries in $Open$
forall the $i \leftarrow 0, \ldots, N! - 1$ do                {initialize $Open$ }
    $Open[i] \leftarrow$ false ;
$Open[rank(\hat{s})] =$ true ;                               {mark initial as open}
while $Open$ has changed do                                   {states marked as open}
    forall the $i \leftarrow 0, \ldots, N! - 1$ do            {scan for states to expand}
        if $Open[i] =$ true then                              {this state expanded or to expand}
            $s \leftarrow unrank(i)$ ;                        {reconstruct state $s$ }
            expand successors $s \rightarrow s_1 \ldots s_\nu$ ;   {generate successors}
            for $s_j$ $(\forall j : 1 \leq j \leq \nu)$ do    {check each successor}
                $Open[rank(s_j)] \leftarrow$ true ;           {mark it for expansion}
```

As this algorithm does not distinguish between open and closed nodes, it may expand a node multiple times. It is able to determine reachable states if a bijective hash function is present. Additional information extracted from a state can improve the running time by decreasing the number of reopened nodes.

If the successor's rank is smaller than the rank of the actual one, it will be expanded in the next scan, otherwise in the same.

Theorem 10 (Number of Scans in 1-Bit Reachability) *The number of scans in the algorithm* One-Bit-Reachability *is bounded by the maximum BFS-Layer.*

Proof. Let $L_b(i)$ be the BFS-Layer and $L_o(i)$ be the layer in the algorithm *One-Bit-Reachability*. Inductively $L_o(i) \leq L_b(i)$ holds. Evidently, $L_o(\mathit{init}) = L_b(\mathit{init}) = 0$. For any path $(s_0, \ldots, s_d)$ generated by BFS, $L_o(s_{d-1}) \leq L_b(s_{d-1})$ holds by induction hypothesis. All successors of s_d are generated in the same iteration (if their index value is larger) or in the next iteration (if their index value is smaller) such that $L_o(s_d) \leq L_b(s_d)$. □

14.1.3 One-Bit Breadth-First search

In cases with the move-alternation property , a BFS can be performed using only one bit per state. In this section the considerations will be exemplified selecting the permutation ordering of Myrvold and Ruskey in the sliding-tile puzzles.

Since the parity does not change in this puzzle another alternating property is needed, and can be found in the position of the blank. The partition into buckets $B_0, \ldots, B_{nm-1}$ enables to determine whether the state belongs to an odd or even layer and which bucket a successor belongs to (Zhou and Hansen, 2004). Numbering the positions from 0 to $nm-1$ and reducing the problem to puzzles with an odd number of columns shows that the successors of a state where the blank is at position 0 will have its blank either at position 1 (right move) or at position n (down move). For puzzles with an even number of columns the position of the blank also indicates the parity of the BFS-Layer. We observe that the blank position in puzzles with an odd number of columns at an even breadth-first level is even and for each odd breadth-first level it is odd.

For such a factored representation of the sliding-tile puzzles, a refined exploration in Algorithm 14.3 retains the breadth-first order, by means that a bit for a node is set for the first time in its BFS-Layer. The bit vector *Open* is partitioned into nm parts, which are expanded depending on the breadth-first *level* (line 7). The directions in which the blank can move (R-right, L-left, D-down,U-up, see line 9), are expanded in parallel using different threads.

Algorithm 14.3: One-Bit-Breath-First search

```
   Input: $\hat{s} \in \mathcal{S}$: initial state, $N! - 1$ number of 2 bit entries in $Open$
1  for blank $= 0, \ldots, nm - 1$ do                {initialize two dimensional $Open$ }
2      for $i = 0, \ldots, (nm - 1)!/2 - 1$ do
3          $Open[blank][i] \leftarrow$ false ;
4  $Open[blank(\hat{s})][rank(\hat{s})$ mod $(nm - 1)!/2] \leftarrow$ true ;
   {mark initial as to expand}
5  $level \leftarrow 0$ ;
6  while $Open$ has changed do                      {states marked as open}
7      $blank \leftarrow level$ mod 2 ;   {expand partitions according to BFS-Level mod2}
8      while blank $\leq nm$ do
9          forall the $d \in \{R, L, D, U\}$ do                     {for all moves}
10             $dst \leftarrow newblank(blank, d)$ ;
11             if $d \in \{L, R\}$ then     {horizontal move, rank does not change here}
12                 $Open[dst] \leftarrow Open[dst]$ or $Open[blank]$ ;
13             else
14                 forall the $i$ with $Open$[blank]$[i] =$ true do
15                     $s \leftarrow unrank(i)$ ;
16                     expand successors $s \rightarrow s_1 \ldots s_\nu$ ;     {generate successors}
17                     for $s_j$ $(\forall j : 1 \leq j \leq \nu)$ do          {check each successor}
18                         $r \leftarrow rank(s_j)$ mod $(N - 1)!/2$ ;
19                         $Open[dst][r] \leftarrow$ true ;
20         $blank = blank + 2$ ;
21     $level = level + 1$ ;
```

As mentioned above, the rank of a permutation does not change by a horizontal move of the blank. This is exploited in line 11 writing the ranks directly to the destination bucket using a bitwise-or on the bit vector from layer *level* $-$ 2 and *level*. The vertical moves are unranked, moved and ranked from line 13 onwards. When a bucket is done, the next one is skipped and the next but one is expanded. The algorithm terminates when no new successor is generated.

Even though some states are expanded several times, the following result is immediate. Let the population count pc_l of level l be the number of bits set after the l-th scan. Then the number of states in BFS-level l is $|\text{Layer}_l| = pc_l - pc_{l-1}$.

14.2 Porting Algorithms to the GPU

When porting the above algorithms to the GPU the specific advantages will be considered one by one. To profit from coalescing, threads should access adjacent memory contemporary. Additionally, the SIMD like architecture forces to avoid if-branches and to design a kernel which will be executed unchanged for all threads. These facts lead to the implementation of keeping the entire or partitioned state space bit vector in RAM and copying an array of indexes (ranks) to the GPU. This approach benefits from the SIMD technology but imposes additional work on the CPU. One additional scan through the bit vector is needed to convert its bits into integer ranks, but on the GPU the work to unrank, generate the successors and rank them is identical for all threads. To avoid unnecessary memory access, the rank given to expand should be overwritten with the rank of the first child. As the number of successors is known beforehand, with each rank space for its successors is reserved. For smaller BFS-Layers this means that a smaller number of states is expanded.

Algorithm 14.4: One-Bit reachability utilizing the GPU

```
Input: $\hat{s} \in \mathcal{S}$: initial state, $N! - 1$ number of 2 bit entries in $Open$
forall the $i \leftarrow 0, \ldots, N! - 1$ do                    {initialize $Open$ }
    $Open[i] \leftarrow$ false ;
$Open[rank(\hat{s})] =$ true ;                                  {mark initial as open}
while $Open$ has changed do                                     {states marked as open}
                    Stage 1 - Generate sets of active transitions
    fillVRAM($\{i | 0 \leq i \leq N! - 1 \wedge Open[i] =$ true$\}$) ;
                                                                {copy ranks to VRAM}
    $Active \leftarrow Active \cup$ GPU-Kernel Determine Transitions() ;
                    Stage 2 - Generate sets of successors
    fillVRAM($Active \cap \leq i \leq N! - 1 \wedge Open[i] =$ true$\}$) ;
                                                                {copy ranks to VRAM}
    $Successors \leftarrow Successors \cup$ GPU-Kernel Generate Successors();
                    Stage 3 - Remove duplicates and adjust $Open$
    forall the $r \in$ Successors do                            {for each rank in Successors}
        $Open[r] \leftarrow$ true ;                             {mark it for expansion}
```

To exemplify the process of porting the algorithms extensions to the *One-Bit reachability* algorithms are given in detail. Algorithm 14.4 exemplifies the modifications to

Algorithm 14.5: *GPU-Kernel* Determine Transitions for One-Bit reachability

```
Input: $\{r_1, \dots, r_k\}$ set of ranks to examine, $T$ set of transitions, $d$ dimension of
       the grid
Output: $\{a_1, \dots, a_k\}$ number of successor for each rank in $\{r_1, \dots, r_k\}$
for each group $g$ do in parallel                    {partially distributed computation}
    for each thread $p : 0 \leq p < d$ do in parallel      {distributed computation}
        $e \leftarrow 0$ ;                                          {reset counter}
        $s \leftarrow unrank(r_{g*d+p})$ ;                {reconstruct state $s$ from rank}
        forall the $t \in T$ do                              {check each transition}
            if $t$ is applicable in $s$ then                   {evaluate transition}
                $e \leftarrow e + 1$ ;                             {increase counter}
        $a_{g*d+p} \leftarrow e$ ;                         {replace rank with number}
return $\{a_1, \dots, a_k\}$ ;                              {return active transitions}
```

Algorithm 14.2 to utilize the GPU for generating states. The algorithm is divided according to the framework given in Part I and the successor counting strategy is used due to no distinction between pre- and post-conditions in the game representation. Since the reversible hash functions *rank* and *unrank* can be computed on the GPU, only the rank is copied over the bus. Obviously, it would be inefficient to copy *Open* as a bit vector to the GPU since a smoothed load balancing in all threads can not be guaranteed.

Algorithms 14.5 and 14.6 present the extended kernels used in the *One-Bit reachability on the GPU*. The extension is based on adding the *rank* and *unrank* functions in the appropriate places. Additionally this kernels accept sets of ranks as the input and the successor generation kernel returns a set of ranks as its output.

In larger instances that exceed RAM capacities additional write buffers are maintained to avoid random access on disk. Once the buffer is full, it is flushed to disk. In one streamed access, all corresponding bits are set.

The setting is exemplified for the sliding-tile puzzle domain in Figure 14.1. We see the "blank-partitioned" breadth-first state space residing on disk that is read into RAM, converted to integer ranks, copied to the GPU to be unranked, expanded and ranked again.

As a surplus, *pthreads* were used as additional multi-threading support. The partitioned state space was divided on multiple hard disks to increase the reading and writing bandwidth and to enable each thread to use its own hard disk.

To sample a move-alternation property in contrast to the blank's position in the sliding-tile puzzles, the Manhattan distance heuristic value has been computed on the GPU by processing the unranked permutation. Even though the estimate can be computed incrementally in constant time, for the sake of generality it was computed from scratch by cumulating absolute distances for all tiles.

The option of computing the heuristic value efficiently on the GPU suggests to also accelerate heuristic search. By the large reduction in state space due to the directedness of the search and by the lack of a perfect hash function for the explored part of the state space in heuristic search – at least for simpler instances – a bit vector compression for

Algorithm 14.6: *GPU-Kernel* Generate Successors for One-Bit reachability

```
Input: $\{r_1, \ldots, r_k\}$ set of ranks to expand, $T$ set of transitions,
       $\{a_1, \ldots, a_k\}$ numbers of successors, $d$ dimension of the grid,
       $Succ$ Successor generation function
Output: Ranks the set of successor ranks
Ranks ← $\emptyset$ ;                                  {clear set of successor ranks}
for each group $g$ do in parallel                     {partially distributed computation}
    for each thread $p : 0 \leq p < d$ do in parallel          {distributed computation}
        $g \leftarrow 0$ ;                             {Counter for generated successors}
        $s \leftarrow$ unrank($r_{g*d+p}$) ;               {reconstruct state $s$ from rank}
        while $g < (a_{g*d+p})$ do                            {generate all successors}
            forall the $t \in T$ do                             {check each transition}
                if $t$ is applicable in $s$ then                 {evaluate transition}
                    Ranks ← Ranks $\cup$ rank($Succ(s,t)$) ;    {add successor rank}
                    $g \leftarrow g + 1$ ;     {increase counter for generated successors}
return Ranks ;
```

the entire search space is not the most space-efficient option. However, as bit vector manipulation is fast, for hard instances runtime advances on the GPU were obtained.

For our case study we have ported breadth-first heuristic search (BFHS) (Zhou and Hansen, 2006) to the GPU. For a given upper bound U on the optimal solution length and current BFS-level g the GPU receives the value $U - g$ as the maximal possible h-value, and marks states with larger h-value as invalid.

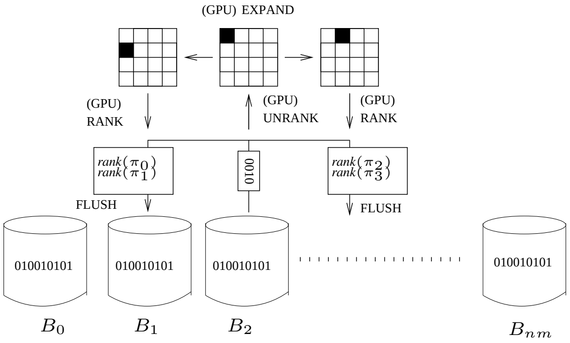


Figure 14.1: GPU Exploration of a Sliding-Tile Puzzle State Space Search stored as a bit vector in RAM.

14.2.1 Case Study: Nine-Men-Morris

Nine-Men Morris is a two-player zero-sum game and the optimality of a move is depending on the player that moves. After generating the state space, the game is solved bottom-up. As the number of pieces in the phases II and III can only decrease, Gasser has applied such retrograde analysis to these two phases. For phase I, however, he applied $\alpha\beta$, which prunes the search space and weakly solves the game. In contrast, this work strongly solves it, and determines the game-theoretical value for each reachable game state.

While for circuit free two-player games 2 bits are sufficient to encode won , loss and draw games and to conduct a retrograde analysis, this game requires a *progress measure* to avoid infinite computations. Gasser's 1 byte encoding is adopted.

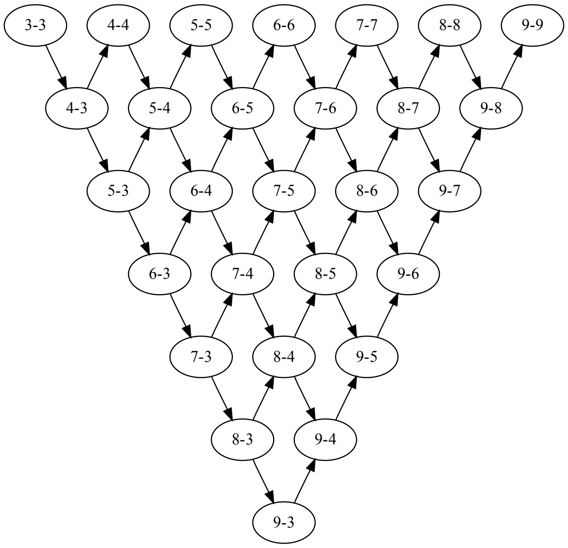


Figure 14.2: Partitioning for Phase II and III.

The partitioning in Fig. 14.2 indicates that for $i, j \in \{1, 2\}$ and $i \neq j$ predecessors of $S_{k_1,k_2,i}$ (Player i to move) are contained either in $S_{k_1,k_2,j}$ or, if player j has closed a mill, in $S_{k_1+1,k_2,j}$ (given $i = 1$) or $S_{k_1,k_2+1,j}$ (given $i = 2$).

Scanning Figure 14.2 from left to right may be interpreted as a variant of space-efficient *frontier search* (Korf, 1999; Korf and Zhang, 2000). To analyze S_{k_1,k_2} with $k_1 > 3$ or $k_2 > 3$ requires the results left to S_{k_1,k_2} to be present. Due to symmetry, S_{k_1,k_2} with $k_1 < k_2$ needs not to be considered again, so that a copy from S_{k_1,k_2} suffices to evaluate a state.

For the encoding of a rank r, 34 bits are sufficient, so that a 64-bit integer suffices to contain all state information. This integer actually stores pairs (r, v) with the additional state information v having 8 bits. The remaining bits are used to store numbers of successors. The GPU expects pairs (r, v) for expansion and returns triples (r', v', c),

Algorithm 14.7: BFS for Phase I

```
1  $Open \leftarrow \emptyset$ ;                                   {initialize $Open$ }
2  for $t \leftarrow 1$ to $4$ do      {closing mills not possible here, so initialization suffices}
3      $Open \leftarrow Open \cup (\lceil \frac{t}{2} \rceil, \lfloor \frac{t}{2} \rfloor, t)$ ;      {mark states in $bits_{\lceil \frac{t}{2} \rceil, \lfloor \frac{t}{2} \rfloor, t}$ as open}
4  for $t \leftarrow 5$ to $18$ do                  {$k_1$ tokens of player 1 and $k_2$ of player 2}
5      forall the $(k_1, k_2, t') \in Open$ with $t' = t - 1$ do
                         Stage 1 and 2 - Generate successors
6          $ranks \leftarrow \emptyset$ ;
7          fillVRAM($\{i | bits_{k_1,k_2,t'}[i] = Open \wedge 0 \leq i < \binom{n}{k_1,k_2}\}$) ;
8          $ranks \leftarrow ranks \cup$ Generate successors in two kernels ;
                      Stage 3 - Remove duplicates and adjust $Open$
9          forall the $r' \in ranks$ do
10             Determine $k'_1, k'_2$ according to $r'$ ;
11             if $(k'_1, k'_2, t) \notin Open$ then
12                 $Open \leftarrow Open \cup (k'_1, k'_2, t)$ ;
13                 Initialize bit vector $bits_{k'_1,k'_2,t}$ with 'not reached' ;
14             Mark $r'$ in $bits_{k'_1,k'_2,t}$ with 'reached' ;
```

where c is the number of successors for the state represented in r' (still fitting into 64 bits). The CPU reads value c if needed for encoding r' more efficiently.

Phase I is not completely analyzed in Gasser (1996) in contrast to this work. Arguing that closing mills is unfortunate, his analysis was reduced to games with 8 or 9 of the 9 pieces for each player. The BFS starts for phase I with an empty board and determines for all depth $t \in \{1, \ldots, 18\}$ which sets $S_{k_1,k_2,t}$ are to be considered and which states are then reached in that set. The partition into sets $S_{k_1,k_2,t}$ is different to the one obtained in the other two phases and respects that some partitions may be encountered in different search depth.

The BFS traversal is shown in Algorithm 14.7. For depths 1 to 4 the state space is initialized with *reached*. Only in depth $t > 4$ closing mills is possible, so that the successors of a set in the two sets of the next depth are possible and, therefore, a growing number of state spaces are to be considered.

The sets are themselves computed in BFS, utilizing a set $Open$ of triples (k_1, k_2, t) with piece counts k_1, k_2 and obtained search depth t. An according entry (k_1, k_2, t) denotes that BFS has reached all states in $S_{k_1,k_2,t}$. This allows to compute the according state spaces incrementally.

If $S_{k_1,k_2,t}$ is encountered for the first time (line 11), prior to its usage in (line 14) the responsible bit vector is allocated and initialized as *not-reached*. In line 10 of the Algorithm 14.7 the outcomes for different k'_1, k'_2 are combined. If depth t is odd we have $k'_1 = k_1 + 1$ but both $k'_2 = k_2$ and $k'_2 = k_2 - 1$ are possible (depending on a mill being closed or not). We take an additional bit in the encoding of the ranks to denote if a mill has been closed to accelerate the determination of values k'_1, k'_2 of rank r'.

In principle, one bit per state is sufficient. Subsequent to the BFS a backward

Algorithm 14.8: Retrograde Analysis Phase I

```
Input: $bits_{k_1,k_2,t}$, $Open$, $bytes_{k_1,k_2,i}$
forall the $(k_1, k_2, t') \in Open\ with\ t' = 18$ do
    if $k_1 \geq k_2$ then                    {symmetry is used to compress Phase I}
        forall the $j \in \{j \mid bits_{k_1,k_2,t}(j) =$ reached $\}$ do
            $bits_{k_1,k_2,t}(j) \leftarrow bytes_{k_1,k_2,1}(j)$ ;
                                {transfer the game theoretical value from Phase II}
    else
        forall the $j \in \{j \mid bits_{k_1,k_2,t}(j) =$ reached $\}$ do
                   {Compute Rank $j'$ of the inverted game state for state with rank $j$}
            $bits_{k_1,k_2,t}(j) \leftarrow bytes_{k_2,k_1,2}(j')$ ;
                                {transfer the game theoretical value from Phase II}
for $t \leftarrow 17$ to $1$ do                                {the remaining layers}
    forall the $(k_1, k_2, t') \in reached\ with\ t' = t$ do
                                Stage 1 and 2 - Generate successors
        $ranks \leftarrow \emptyset$ ;
        fillVRAM($\{j|bits_{k_1,k_2,t'}[j]$ =reached $\wedge\ 0 \leq i < \binom{n}{k_1,k_2}\}$) ;
        $ranks \leftarrow ranks\ \cup$ Generate successors in two kernels ;
                         Stage 3 - Remove duplicates and adjust $Open$
        forall the $r' \in ranks$ do
            Compute $k'_1, k'_2$ associated with $r'$ ;
            $bits_{k_1,k_2,t}(r) \leftarrow bits_{k'_1,k'_2,t+1}(r')$ ;
```

chaining algorithm determines the game-theoretical values. Two bits are used per state to encode the four cases *not-reached*, *won-for-player-1*, *won-for-player-2*, and *draw*. As already 1 bit is used for state-space generation, these demands are already allocated.

In the backward traversal described in Algorithm 14.8, first all state sets $S_{k_1,k_2,t}$ with depth $t = 18$ are initialized wrt. the data computed for phase II and III. As player 1 starts the game, he will also start phase II. For the initialization of $S_{k_1,k_2,t}$ with $t = 18$ and $k_1 \geq k_2$ we scan the corresponding bit vector and consider each state marked *reached* at position i the value stored with position i in the byte vector inferred for $S_{k_1,k_2,1}$ from solving phase II and III. Depth and successor count information is ignored. We are only interested, whether a state is won, lost or a draw.

When trying to initialize $S_{k_1,k_2,t}$ with $t = 18$ and $k_1 < k_2$ we observe that no corresponding set $S_{k_1,k_2,1}$ from phase II and III has been computed. In this case, we traverse the bit vector for $S_{k_1,k_2,t}$, but consider the set $S_{k_2,k_1,2}$ from phase II and III. In each scan of $S_{k_1,k_2,t}$ when encountering a state s marked *reached* we compute the rank j of its inverted representation, so that player 1 now plays color 2 and player 2 plays color 1. Similarly, in case $S_{k_1,k_2,1}$ is not present, we consider position j in the byte vector, while inverting the state to be considered. For the translation of states in state vector representation, their inverted representation and the computation of their ranks j, we use the GPU.

After the initialization one step back is performed and the work continues on the state spaces $S_{k_1,k_2,t}$ with $t = 17$. All reached successors of a state are generated and the game-theoretical value is determined in the bit vector stored depth 18 by considering all values at positions that correspond to the successor ranks.

The value of a state is determined by considering all its successors. If all successors of a state with player 1 to move are lost, the state itself is lost. If at least one successor is won, then the state itself is won. In all remaining cases, the game is a draw. We continue until we reach depth 1.

14.3 Summary

Having introduced three algorithms to enumerate state spaces of games by using only two or even one bit internal memory per state a porting of this algorithms to the GPU is proposed. A special case, described in detail was the two player game Nine-Men-Morris which was solved according to Gasser's approach accelerated by a GPU.

Chapter 15

Experimental evaluation

The introduced games are compared to a sequential CPU implementation. The next sections will give distinct results of the evaluation. For measuring the speed-up on a matching implementation the GPU performance is compared with a CPU emulation on a single core. This way, the same code and work was executed on the CPU and the GPU. The emulation was run with one thread to minimize the work for thread communication on the CPU.

Structure of the chapter: In the next sections the state enumeration algorithms ported to the GPU are evaluated on different games, followed by details on the solution of Nine-Men-Moris.

15.1 Single-Agent Games

First the behavior of the 1-bit BFS was examined and the GPU to the CPU time compared. Since calculating large binomial coefficients is a computationally intensive task, values were precomputed and stored in an array.

Table 15.1: GPU vs. CPU Performance using 1-Bit BFS (o.o.t. denotes out of time).

Domain	Instance	Times		
		GPU	CPU	CPU
		GPU	1 Core	8 Cores
Sliding-Tile	(3×4)	$66s$	$427s$	$217s$
	(4×3)	$78s$	$475s$	$187s$
	(2×6)	$93s$	$1,114s$	$374s$
	(6×2)	$114s$	$1,210s$	$284s$
	(7×2)	$14,215s$	o.o.t.	$22,396$
Peg-Solitaire		$44s$	$360s$	

Table 15.2: GPU vs. CPU Performance using 2-Bit BFS.

Domain	Instance	Times		
		GPU	CPU 1 Core	CPU 8 Cores
Top-Spin	10	$0s$	$2s$	$0s$
	11	$1s$	$10s$	$3s$
	12	$12s$	$272s$	$63s$
	13	$87s$	$2,404s$	$510s$
Pancake	10	$0s$	$4s$	$2s$
	11	$9s$	$52s$	$14s$
	12	$130s$	$832s$	$164s$
	13	$1,819s$	$11,771s$	$2,499s$
Frogs and Toads		$686s$	$8,880s$	

The first set of experiments in Table 15.1 shows the gain of integrating bit vector state space compression with BFS in different instances of the sliding-tile puzzle. We run the one-bit BFS algorithm on various instances of the sliding-tile-puzzle with RAM requirements from 57 MB up to 4 GB. The 3×3 version was simply too small to show significant advances, while even in partitioned form a complete exploration on a bit vector representation of the 15-Puzzle requires more RAM than available. Moreover, the predicted amount of 1.2 TB hard disk space is only slightly smaller than the 1.4 TB of frontier BFS search reported by Korf and Schultze (2005).

For the 1-Bit BFS implementation the speed-up achieves a factor between 7 and 10 in the small instances. Many states are re-expanded in this approach, inducing more work for the GPU and exploiting its potential for parallel computation. Partitions being too large for the VRAM are split and processed in chunks of about 250 millions indexes (for the 7×2 instance). A quick calculation shows that the savings of GPU computation are large. It should be noted that the GPU has the capability to generate 83 million states per second (including unranking, generating the successors and computing their ranks) compared to about 5 million states per second of the CPU (utilizing one core). As a result, for the CPU experiment that ran out of time (o.o.t.), stopped after one day of execution, a speed-up factor of at least 16, and a running time of over 60 hours can be assumed. We also implemented a multi-core version of the algorithm utilizing the available 8 cores and showing the benefit of the GPU implementation.

The results for the (n, k)-Top-Spin problems for $k = 4$ are shown in Table 15.2. Since no layer-selection or move-alternating property exists, 2-bit BFS is performed. Additionally the GPU performance to a parallel CPU implementation is compared. For large values of n, a significant speed-up of more than factor 27 wrt. the single core computation was obtained, and a factor of 5 compared to the 8 core computation.

Table 15.2 depicts the GPU and CPU running time results for the n-Pancake problems. In contrast to the Top-Spin puzzle for a large value of n, a speed-up factor of 7 wrt. running the same algorithm on one core of the CPU is obtained.

15.2 Nine-Men-Morris

Table 15.3: Retrograde Analysis of middle and ending stage in Nine-Men-Morris (times in seconds, sizes given in GB).

	Size	GPU	CPU	Ratio		Size	GPU	CPU	Ratio
9-9	6.54	36,598s	–	–	**8-4**	**1.34**	**610s**	**8,093s**	**13.26**
9-8	8.41	57,057s	–	–	**9-3**	**0.58**	**807s**	**8,547s**	**10.59**
8-8	9.47	35,441s	–	–	6-5	1.15	493s	1,484s	3.01
9-7	8.41	43,003s	–	–	**7-4**	**0.80**	**397s**	**4,434s**	**11.17**
8-7	8.41	61,750s	–	–	**8-3**	**0.40**	**609s**	**6,123s**	**10.05**
9-6	6.54	12,174s	–	–	5-5	0.48	11s	23s	2.09
7-7	6.73	15,284s	52,441s	3.43	6-4	0.40	157s	439s	2.79
8-6	5.89	19,538s	57,988s	2.96	7-3	0.23	357s	3,145s	8.80
9-5	**3.93**	**2,045s**	**25,134s**	**12.29**	5-4	0.16	5s	6s	1.20
7-6	4.28	4,914s	22,981s	4.98	6-3	0.11	69s	619s	8.97
8-5	**3.21**	**1,805s**	**20,257s**	**11.22**	4-4	0.05	1s	1s	1.00
9-4	**1.78**	**829s**	**10,725s**	**12.93**	5-3	0.04	17s	137s	8.05
6-6	2.50	1,137s	4,160s	3.62	4-3	0.01	3s	26s	8.66
7-5	2.14	1,211s	11,682s	9.64	3-3	.003	12s	80s	6.66

The time and space performance of the retrograde analysis is shown in Table 15.3. Since 24 GB were not sufficient to maintain all responsible sets in RAM, states were sequentially flushed to (and subsequently read from) disk. The entire classification of the state space for the middle and ending stage on the GPU required about 3 days and 19 hours. The corresponding CPU computation on one core has not been completed and has been terminated after 5 days.

The observed speed-ups of over one order of magnitude have been obtained (plotted in bold font), exceeding the number of cores on most current PCs. Note that this assertion is true for the dual 6-core CPUs available from Intel, but not on a dual Xeon machine with two quad-core CPUs creating 16 logical cores due to multi-threading. Nonetheless, better speed-ups are possible since NVIDIA GPUs can be used in parallel and the Fermi architecture (e.g. located on the GeForce GTX 480 graphics card) is coming out which will go far beyond the 240 GPU cores we had access to.

For larger levels, however, we observe that the GPU performance degrades. When profiling the code, we identified I/O access as one limiting factor. For example, reading $S_{8,8}$ from one HDD required 100 seconds, while the expansion of 8 million states, including ranking and unranking required only about 1 second on the GPU.

Nonetheless still some inconsistencies in the GPU performance can be observed. According to the calculations, due to storing intermediate results, 24 GB RAM should be sufficient for the bit vector for BFS and retrograde analysis in the RAM, so that no further access to HDD for swapping should have been necessary. But there is additional memory needed for preparing and postprocessing the VRAM in RAM for copy

Table 15.4: Retrograde analysis in opening stage in Nine-Men-Morris (times in seconds).

Depth	BFS	Retrograde	Depth	BFS	Retrograde
1	<1s	<1s	10	171s	163s
2	<1s	<1s	11	390s	388s
3	<1s	<1s	12	909s	885s
4	<1s	<1s	13	1,583s	1,554s
5	<1s	<1s	14	2,838s	2,828s
6	1s	1s	15	4,047s	4,021s
7	4s	4s	16	5,743s	5,996s
8	17s	17s	17	7,219s	6,996s
9	53s	52s	18	-	16,141s

purposes. Together with the needs of the operating system this indicated that the system did swap at least to some extend.

For analyzing the opening stage the program required little less than 17 hours. Table 15.4 depicts the individual timings obtained by the GPU. All 24 states in depth 1 turned out to be a draw such that the result of Gasser (1996) has been validated. First non-optimal moves are possible in depth 2.

15.3 Summary

Having introduced the analyzed games the first chapter of this part continues with an introduction to Game Solving. Due to the simple structure of the identified games perfect hash functions are introduced divided in *ranking* and *unranking* for permutation games with distinguishable pieces, and *binomial* and *multinomial* hashing for one and more player games with indistinguishable pieces. Chapter 14 proposes a set of algorithms suitable to enumerate the state space by traversing it in a BFS manner and ports the algorithms to the GPU making use of the framework proposed in Part I. A special case is the two player game Nine-Men-Morris. Here the solving process includes a backward search called retrograde analysis. This Part closes with a chapter showing the experimental results obtained while comparing the GPU approaches to a single and a parallel CPU implementation. Significant speed-ups were achieved with a speed-up factor up to 27 compared to the single CPU implementation.

Part V

Probabilistic Model Checking

Chapter 16

Introduction Probabilistic Model Checking

The fourth discipline to analyze is Probabilistic Model Checking. Here a pure state space search has shown to be inefficient (Kwiatkowska *et al.*, 2004) so a numerical strategy is used. Probabilistic Model Checking (Bosnacki *et al.*, 2009) boils down to solving linear equations via computing multiple sparse matrix-vector products. The mathematical background is parallelizing Jacobi iterations. While the PCTL Probabilistic Model Checking approach accelerates one iterated numerical operation on the GPU, for explicit-state LTL Model Checking a single scan over a large search space is performed. As a result, this work proposes a conceptually different algorithm, suited to parallel Model Checking of large models in Probabilistic Model Checking.

Structure of the chapter: This chapter introduces the basics of the Probabilistic Model Checking analogous to Kwiatkowska *et al.* (2007) and Baier and Katoen (2008). The introduction mainly focuses on discrete-time Markov chains (DTMCs) and the logic PCTL, and discusses only briefly continues-time Markov chains. It will motivate the usage of matrix vector multiplication and solving systems of linear equations being the essential aspects of nearly all algorithms for Probabilistic Model Checking.

16.1 Discrete Time Markov Chains

Given a fixed finite set of atomic propositions AP a DTMC is defined as follows:

Definition 38 *A (labeled) DTMC $\mathcal{D}$ is a tuple $(S, \hat{s}, \mathbf{P}, L)$ where*

- *$\mathcal{S}$ is a finite set of states;*
- *$\hat{s} \in S$ is the initial state;*
- *$\mathbf{P} : S \times S \rightarrow [0, 1]$ is the transition probability matrix where $\Sigma_{s' \in S}\mathbf{P}(s, s') = 1$ for all $s \in S$;*

- $L : S \to 2^{AP}$ *is a labeling function which assigns to each state* $s \in S$ *the set* $L(s)$ *of atomic propositions that are valid in the state.*

In Probabilistic Model Checking the guard of a transition is replaced by a probability given by a real number $\mathbf{P}(s, s')$ in the interval $[0, 1]$ and ensured that for each state the sum of the probabilities of all outgoing transitions sum up to 1 and, consequently, each state without outgoing edges is extended by a self loop with the probability of 1.

16.2 Probabilistic Computational Tree Logic

Properties of DTMCs can be specified using *Probabilistic Computation Tree Logic (PCTL)* (Hansson and Jonsson, 1994), which is a probabilistic extension of CTL.

Definition 39 *PCTL has the following syntax:*

$$\Phi = \texttt{true} \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid P_{\sim p}[\phi]$$

$$\phi = \texttt{X}\ \Phi \mid \Phi\ \texttt{U}\ ^{\leq k}\Phi$$

where $a \in AP$, $\sim \in \{<, \leq, \geq, >\}, p \in [0, 1]$, *and* $k \in \mathbb{N} \cup \{\infty\}$.

The above definition features both state formulas Φ and path formulas ϕ, which are interpreted on states and paths, respectively, of a given DTMC $\mathcal{D}$. However, the properties are specified exclusively as state formulas. Path formulas have only an auxiliary role and they occur as a parameter in state formulas of the form $P_{\sim p}[\phi]$. Intuitively, $P_{\sim p}[\phi]$ is satisfied in some state s of $\mathcal{D}$, if the probability of choosing a path that begins in s and satisfies ϕ is within the range given by $\sim p$. To formally define the satisfaction of the path formulas one defines a probability measure, which description is beyond the scope of this introduction. Informally, this measure captures the probability of taking a given finite path in the DTMC, which is calculated as the product of the probabilities of individual transitions of this path.

The path operators have intuitive meaning which is analogous to the one in standard temporal logics. The formula $\texttt{X}\ \Phi$ is true if Φ is satisfied in the next state of the path. The bounded until formula $\Phi\ \texttt{U}\ ^{\leq k}\Psi$ is satisfied if Ψ is satisfied in one of the next k steps and Φ holds until this happens. For $k = \infty$ one obtains the unbounded until. In this case the superscript is omitted and $\Phi\ \texttt{U}\ \Psi$ used. The interpretation of unbounded until is the standard strong until. Figure 16.1 exemplifies a probabilistic model representing the thesis problem given in the introduction.

16.3 Algorithms for Model Checking PCTL

Given a labeled DTMC $\mathcal{D} = (S, \hat{s}, P, L)$ and a PCTL formula Φ, an algorithm verifies whether the initial state of $\mathcal{D}$, $\hat{s}$, satisfies Φ.

Nevertheless, the algorithm works by checking the satisfaction of Φ for each state in $\mathcal{S}$. The output of the algorithm is $Sat(\Phi)$, the set of all states that satisfy Φ.

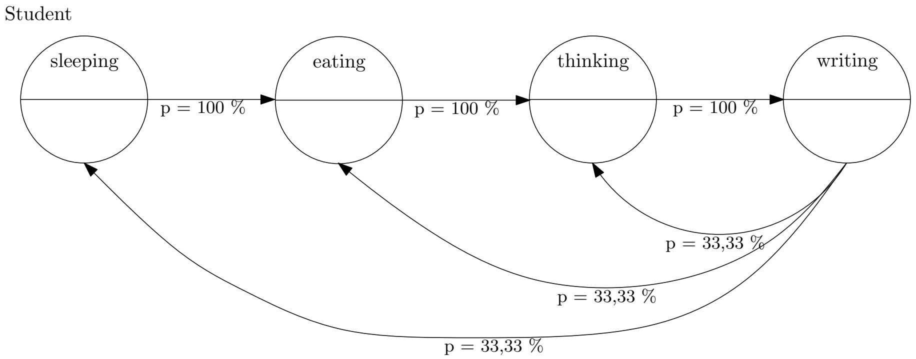


Figure 16.1: Thesis problem (Figure 1.1) as a probabilistic graph.

The algorithm starts by first constructing the parse tree of the PCTL formula Φ. The root of the tree is labeled with Φ and each other node is labeled by a sub formula of Φ. The leaves are labeled with `true` or an atomic proposition. Starting with the leaves, in a recursive bottom-up manner for each node n of the tree the set of states is computed that satisfies the sub formula that labels n. When it arrives at the root $Sat(\Phi)$ can be determined.

Model checking algorithms for the state PCTL formulas are analogous with their counterparts in CTL and as such quite straightforward to implement. The only exceptions are the path formulas whose algorithms contain an extensive numerical component that is used to compute the transition probabilities. They are the most computationally demanding part of the Model Checking algorithm and as such a logical target of the improvement via parallel algorithms for GPUs.

To get a general picture above these claims consider the algorithm for the formulas of the form $\mathtt{P}[\Phi \ \mathtt{U}^{\leq k}\Psi]$, where $k = \infty$. The numerical component of this case reduces to finding the least solution of the linear equation system:

$$\mathbf{W}(s, \Phi \mathtt{U} \Psi) = \begin{cases} 1 & \text{if } s \in Sat(\Psi) \\ 0 & \text{if } s \in Sat(\neg\Phi \wedge \neg\Psi) \\ \Sigma_{s' \in S}\mathbf{P}(s, s') \cdot \mathbf{W} & (s', \Phi \mathtt{U} \Psi) \text{ otherwise} \end{cases}$$

where $\mathbf{W}(\Phi \ \mathtt{U} \ \ \Psi)$ is the resulting vector of probabilities indexed by the states in $\mathcal{S}$. Only the states in which the formula is satisfied with probabilities 1 and 0 have a special treatment. For each other state the probabilities are computed in a recurrent fashion using the corresponding probabilities of the neighboring states. Before solving the system, the algorithm employs some optimizations by precomputing the states that satisfy the formula with probability 0 or 1. The (simplified) system of linear equations can be solved using iterative methods that comprise matrix-vector multiplication. One such method is presented by Bronshtein and Semendyayev (1997), which is also one of the methods that PRISM uses and which is described in more detail in Chapter 17. Jacobi's method is preferred in this work over other methods that usually perform better on sequential architectures. This is because Jacobi has certain advantages in the parallel programming context. For instance, it has lower memory consumption compared to the Krylov subspace methods (Nevanlinna, 1993) and less data dependencies than the Gauss-Seidel method (Jeffreys and Jeffreys, 1988), which makes it easier to parallelize (Bell and Haverkort, 2006). The algorithms for the next operator and bounded until boil down to a single matrix-vector product and a sequence of such products, respectively.

PCTL can be extended with various rewards ($cost$) operators that are not given here. The algorithms for those operators can also be reduced to matrix-vector multiplication (Kwiatkowska *et al.*, 2007).

Thus, the main runtime bottleneck of the probabilistic Model Checking algorithms is the computational part, and in particular the linear algebraic operations. Their share of the total runtime of the algorithms increases with the size of the model $|\mathcal{S}|$. Model checking of a PCTL formula Φ on DTMC $\mathcal{D}$ is linear in $|\Phi|$, the size of the formula, and polynomial in $|\mathcal{S}|$, the number of states of the DTMC. The most expensive are the operators for unbounded until and also the rewards operators which too boil down to solving system linear equations of size at most $|\mathcal{S}|$. The complexity is linear in k_{max}, the maximal value of the bounds k in the bounded until formulas (which also occurs in some of the costs operators). However, usually this value is much smaller than $|\mathcal{S}|$. So, for real world problems, that tend to have large state spaces, the dependence on the size $|\mathcal{S}|$ is even more critical.

16.4 Beyond Discrete Time Markov Chains

Matrix-vector product is also in the core of Model Checking continuous-time Markov chains, i. e., the corresponding Computational Stochastic Logic (CSL) (Kwiatkowska *et al.*, 2007; Baier *et al.*, 2003; Bell and Haverkort, 2006). For instance, the next operator of CSL can be checked in the same way like its PCTL counterpart. Both algorithms for steady state and transient probabilities boil down to matrix-vector multiplication. On this operation hinge also various extensions of CSL with costs. Thus, the parallel version of the Jacobi algorithm presented in the sequel, can also be used for stochastic models, i. e., models based on CTMCs.

16.5 Summary

Having defined the input language for probabilistic models the chapter continues with an introduction to algorithms used in this domain. In the following the application of GPUs to accelerate the solution finding given a set of linear equations is described.

Chapter 17

GPU Enhanced Probabilistic Model Checking

To speed up the algorithms the sequential matrix-vector multiplication algorithm is replaced with a parallel one, which is adapted to run on the GPU. This section describes the parallel algorithms which are derived from the Jacobi algorithm for matrix-vector multiplication first published in (Bosnacki *et al.*, 2009) and extended to multiple GPUs in (Bosnacki *et al.*, 2011). This algorithm is used for both bounded and unbounded until, i.e., also for solving systems of linear equations.

Structure of the chapter: The chapter introduces the usage of Jacobi iterations to solve probabilistic problems. Each iteration boils down to a matrix vector multiplication so efficient representations of matrices in the GPU are studied. In the following an approach to parallelize the Jacobi method for GPUs is presented and extended to multiple GPUs.

17.1 Jacobi Iterations.

As mentioned in Chapter 16 for Model Checking DTMCs, Jacobi iteration method is one option to solve the set of linear equations derived for until (`U`). Each iteration in the Jacobi algorithm involves a matrix-vector multiplication. Let $n = |\mathcal{S}|$ be the size of the state space, which determines the dimension $n \times n$ of the matrix to be iterated.

The formula of Jacobi for solving $Ax = b$ iteratively for an $n \times n$ matrix $A = (a_{ij})_{0 \leq i,j \leq n-1}$ and a current vector x^k is

$$x_i^{k+1} = 1/a_{ii} \cdot \left(b_i - \sum_{j \neq i} a_{ij} x_j^k \right), \text{ for } i, j \in \{0, \ldots, n-1\}.$$

For better readability (and faster implementation), one may extract the diagonal elements and invert them prior to applying the formula. Setting $D_i = 1/a_{ii}$, $i \in$

$\{0, \ldots, n-1\}$ then yields

$$x_i^{k+1} = D_i \cdot \left(b_i - \sum_{j \neq i} a_{ij} x_j^k \right), \text{ for } i, j \in \{0, \ldots, n-1\}. \tag{17.1}$$

The sufficient condition for Jacobi iteration to converge is that the magnitude of the largest eigenvalue (spectral radius) of matrix $D^{-1}(A-D)$ is bounded by value 1. Fortunately, the Perron–Frobenius theorem asserts that the largest eigenvalue of a (strictly positive) stochastic matrix is equal to 1 and all other eigenvalues are smaller than this value, so that $\lim_{k\to\infty} A^k$ exists. In the worst case, the number of iterations can be exponential in the size of the state space, but in practice k, the number of iterations until conversion to some sufficiently small ϵ according to a termination criteria, like $\max_i |x_i^k - x_i^{k+1}| < \epsilon$, is often moderate (Stewart, 1994).

17.2 Sparse Matrix Representation.

The size of the matrix being $\Theta(n^2)$ is usually compressed due to the sparsity of the models. Such a matrix compaction is a standard technique used for probabilistic Model Checking and to this end special structures are used. In the algorithms presented here the so called *modified compressed sparse row/column format* (Bell and Haverkort, 2006) is assumed.

Table 17.1: Non-zero elements of a sparse matrix $\mathbf{P}$. The array labeled with *row* and *col* contain the indexes of the non-zero value given in the array *non-zero*.

row	0	0	0	1	1	2	2	2	3	4	4
col	1	2	4	2	3	0	3	4	0	0	2
non-zero	0.2	0.7	0.1	0.01	0.99	0.3	0.58	0.12	1.0	0.5	0.5

The matrix given in Table 17.1 contains in the *non-zero* array the non-zero elements of $\mathbf{P}$ whose row and column indexes are given in the arrays labeled with *row* and *col*. Formally, for all r of the index range of the arrays, $\textit{non-zero}_r = \mathbf{P}(\textit{row}_r, \textit{col}_r)$. This, already optimized format, representing the standard full matrix $\mathbf{P}$, can be compressed further by replacing the *row* array by an array denoting the number of elements in each array. The resulting representation is displayed in Table 17.2 which, in fact, is the mentioned modified compressed sparse row/column format.

Instead of the row indexes, the array *rsize* contains the row sizes, i. e., $\textit{rsize}_i$ contains the number of non-zero elements in row i of $\mathbf{P}$. To extract row i of the original matrix $\mathbf{P}$, take the elements

$$\textit{non-zero}_{\textit{rstart}_i}, \textit{non-zero}_{\textit{rstart}_i+1}, \ldots, \textit{non-zero}_{\textit{rstart}_i+\textit{rsize}_i-1}$$

where $\textit{rstart}_i = \sum_{k=0}^{i-1} \textit{rsize}_k$.

Table 17.2: Modified compressed sparse row/column representation of $\mathbf{P}$, given in Table 17.1. The array *rsize* denotes the number of non-zero elements in the corresponding array.

rsize	3	2	3	1	2						
col	1	2	4	2	3	0	3	4	0	0	2
non-zero	0.2	0.7	0.1	0.01	0.99	0.3	0.58	0.12	1.0	0.5	0.5

17.3 Algorithm Implementation.

The pseudo code of the sequential Jacobi algorithm that implements the aforementioned recurrent formula and which uses the compression given above is shown in Algorithm 17.1.

The **While** loop in line 3 is repeated until the sufficient precision for a termination is achieved or, to ensure a termination, a maximal number of iterations, defined by the user in $\mathtt{max}_k$ is reached. In line 6 an element of the vector b is copied directly into the result vector and the first (f) and last (l) element of the *non-zero* for result x_i^{k+1} is computed. The **ForAll** loop starting in line 9 computes the product of row i and the result of the previous iteration, vector x^k. Afterwards the result is multiplied with the entry in the diagonal array (line 11).

The first optimization, already implemented in PRISM, is to omit intermediate results by using only two vectors x and x' to store x^k the multiplication vector and x^{k+1} the current result. After each iteration the contents of the vectors are swapped for the next iteration. This observation is also used in the parallel algorithm to save space, here, in fact, only the links representing the vectors are swapped. This can be compared to just renaming the vectors and is more efficient then copying the data.

In line 12 the algorithm assumes that a sufficient precision has been reached and checks if one of the pairs x_i^k and x_i^{k+1} violates this assumption. If so the *Terminate* variable is set to `false` to force another iteration. Note, stopping this loop after the first violation would be possible to prevent a complete scan.

Due to the fact that the iterations have to be performed sequentially the matrix-vector multiplication is the part to be distributed. As a feature of the algorithm (that contributed most to the speedup) the comparison of the two solution vectors, x and x' in this case, is done in parallel. The GPU version of the Jacobi algorithm is given in Algorithms 17.2 and 17.3.

Before the computation can start the matrix data is copied to the GPU memory (VRAM) and space for the vector x' and the *Terminate* variable is allocated. Since the *rsize* array contains only the sizes of the distinct columns it has to be converted. Otherwise every thread would have to traverse the array and sum up all entries up to the needed location. The *rsize* array is converted to a *rstart* array in the **For** loop staring in line 5 by storing the sum of the last entry in *rstart* and *rsize* as the following entry of *rstart*. After this conversion the array *rstart* contains at the position *rstart*$_i$ the index of the first entry of column i in the *non-zero* array. This conversion is not done on the GPU since it is a pure sequential task and here the CPU is preferred. The *rstart* array

Algorithm 17.1: Jacobi iteration with row compression (as implemented in PRISM)

```
Input: $b$, $rsize$, non-zero, col, $D$: data representing matrix $\mathbf{P}$,
$\max_k$: maximal number of iterations,
$x$: vector for multiplication
Output: $x$: resulting vector of the multiplication
$k \leftarrow 0$ ;                                   {counter for the iterations}
Terminate $\leftarrow$ false ;                       {helper variable for forced termination}
while (not Terminate and $k < \max_k$) do
{loop until termination or until the maximum number of iterations is reached}
    $l \leftarrow 0$ ;                               {start with the first non-zero entry}
    for $i \leftarrow 0 \dots n$ do                  {traverse all rows}
        $x_i^{k+1} \leftarrow b_i$ ;                 {copy an element into the resulting vector}
        $f \leftarrow l$ ;                           {first entry of row $i$ in the col array}
        $l \leftarrow f + rsize_i - 1$ ;             {last entry of row $i$ in the col array}
        for $j \leftarrow f \dots l$ do              {traverse all entries in the col and non-zero arrays}
            $x_i^{k+1} \leftarrow x_i^{k+1} - \left(\textit{non-zero}_j \cdot x_{col_j}^k\right)$
            {compute the intermediate result}
        $x_i^{k+1} \leftarrow x_i^{k+1} \cdot D_i$ ; {multiply it with the entry in the diagonal array}
    Terminate $\leftarrow$ true ;                    {assume all new entries meet the range}
    forall the $i \leftarrow 0 \dots n$ do           {traverse all rows}
        if $|x_i^{k+1} - x_i^k| > \epsilon$ then     {if the new value is beyond the range}
            Terminate $\leftarrow$ false ;           {force another iteration}
    $k \leftarrow k + 1$ ;                           {increase the iteration counter}
```

is then copied to the GPU end denoted by *rstartGPU* in line 7.

When calling a CUDA kernel the dimension of the grid, meaning number of blocks and the block size, is defined enclosed by <<< and >>>. The algorithm determines the number of blocks by dividing n with a *BlockSize*, defined by the user and increasing the result by 1. Line 2 in Algorithm 17.3 ensures that threads in the last block are terminated if their index exceeds n. After the kernel execution the value of the *TerminateGPU* variable is copied back to the host memory, k is increased and the loop eventually started again. This copy statement serves also as a synchronization barrier, since the CPU program waits until all the threads of the GPU kernel have terminated before copying the variable from the GPU global memory. If another iteration is needed x and x' are swapped[1]. After all iterations the result is copied back from global memory to RAM.

JacobiKernel shown in Algorithm 17.3 is the so-called kernel that operates on the GPU. Local variables d, l, h, i and j are located in the local registers and they are not shared between threads. The other variables reside in the global memory. The result

[1] Since C operates on pointers, only these are swapped in this step.

Algorithm 17.2: Host part of the Jacobi iteration, for unbounded until

```
   Input: $x$ vector to multiply, $n$ number of entries in $x$,
   Diag, $b$, rsize, non-zero, col matrix $\mathbf{P}$,
   $\epsilon$ upper bound for distance, $\max_k$ upper bound for iterations
   Output: $x'$ resulting vector, $k$ number of iterations
1  allocate global memory for x' ;
2  allocate global memory for col, non-zero, b, x, $\epsilon$, $n$ and copy them ;
3  allocate global memory for TerminateGPU to be shared between blocks ;
4  $rstart_0 \leftarrow 0$ ;                                  {Converted rsize array}
5  Terminate $\leftarrow$ false ;             {helper variable for immediate termination}
6  for $i \leftarrow 1 \ldots |\mathrm{rsize}| + 1$ do          {for each entry in the rstart array}
7      $rstart_i \leftarrow rstart_{i-1} + rsize_{i-1}$ ;
       {store the sum of the previous rsize entry and the current rstart array}
8  allocate global memory for rstartGPU and copy rstart to rstartGPU ;
9  $k \leftarrow 0$ ;                                      {counter for the iterations}
10 while (not Terminate and $k < \max_k$) do
   {loop until termination or until the maximum number of iterations is reached}
11     <<<n/BlockSize+1,BlockSize>>>JacobiKernel() ;        {start the kernel}
12     copy TerminateGPU to Terminate ;            {copy terminating information}
13     Swap(x,x') ;          {swap links to previous result and the current result}
14     $k \leftarrow k + 1$ ;                           {increase counter for iterations}
15 copy x' to RAM ;                                        {copy result from GPU}
```

is first computed in d (locally in each thread) and then written to the global memory (line 11). This approach minimizes the access to the global memory from threads. At invocation time each thread computes the row i of the matrix that it will handle. This is feasible because each thread knows its *ThreadId*, and the *BlockId* of its block. Note that the size of the block (*BlockSize*) is also available to each thread. Based on value i only one thread (the first one in the first block) sets the variable *TerminateGPU* to true. Recall, this variable is located in the global memory, and it is shared between all threads in all blocks. Now, each thread reads three values from the global memory (line 5 to 7), here we profit from coalescing done by the GPU memory controller. It is able to detect neighboring VRAM access and combine it.

This means, if thread i accesses 2 bytes at b_i and thread $i+1$ accesses 2 bytes at b_{i+1} the controller fetches 4 bytes at b_i and divides the data to serve each thread its chunk. In each iteration of the for loop an elementary multiplication is done. Due to the compressed matrix representation a double indirect access is needed here. As in the original algorithm the result is multiplied with the diagonal value D_i and stored in the new solution vector x'. Finally, each thread checks if another iteration is needed and consequently sets the variable *TerminateGPU* to `false`. Concurrent writes are resolved by the GPU memory controller. The implementation in Algorithm 17.2 matches the one for bounded-until ($\mathtt{U}^{\leq \mathtt{k}}$), except that bounded-until has a fixed upper bound on the number of iterations, while for until a termination criterion applies.

Algorithm 17.3: *GPU-Kernel* Jacobi iteration with row compression

```
Input: $x$ vector for multiplication, $n$ number of entries in $x$,
Diag, $b$, rstartGPU, non-zero, col matrix $\mathbf{P}$,
$\epsilon$ maximal distance of vector entries
Output: $x'$ resulting vector, TerminteGPU termination result
$i \leftarrow BlockId \cdot BlockSize + ThreadId$ ;
{compute the array index for this thread}
if $(i = 0)$ then                        {first thread of all resets the terminate variable}
    TerminateGPU $\leftarrow$ true ;             {assume this is the last iteration}
if $(i < n)$ then
    $d \leftarrow b_i$ ;                     {temporary variable for intermediate results}
    $l \leftarrow rstartGPU_i$ ;                          {first element of the array}
    $h \leftarrow rstartGPU_{i+1} - 1$ ;                    {last element of the array}
    forall the $j \leftarrow l \ldots h$ do                          {visit all elements}
        $d \leftarrow d - non\text{-}zero_j \cdot x_{col_j}$ ;       {compute the intermediate result}
    $d \leftarrow d \cdot Diag_i$ ;           {multiply it with the entry in the diagonal array}
    $x'_i \leftarrow d$ ;                             {store the result in the vector}
    if $|x_i - x'_i| > \epsilon$ then                {check distance for this threads results}
        TerminateGPU $\leftarrow$ false ;        {force another iteration if necessary}
```

17.4 Extending the Algorithm to Multiple GPUs

The limiting factor when utilizing the graphics card for probabilistic Model Checking is the amount of available global memory. Since the whole matrix needs to be stored on the GPU the approach needs to be extended efficiently to utilize all available cards in the system.

When splitting the data among the cards the multiplication has to be divided and all cards have to be synced after each iteration. An approach to parallelize the iteration fails due to their sequential execution and the motivation to distribute the data. Analyzing Formula 17.1 yields to the result that x^k has to be available on each card, but the matrix $\mathbf{P}$ can be distributed by rows. Although this approach is not scalable to very large instances, since the vector x has to fit into the GPU memory, the currently available maximal number of 4 GPUs in a system is restricting the possibilities earlier. However, distributing x^k after each iteration will increase the amount of the transferred data between the GPU and the host significantly and decrease the efficiency of the approach.

Assume D being the number of cards in the system, the resulting vector is divided in D partitions of size $\lfloor n/D \rfloor$ and $\lceil n/D \rceil$ for the last partition. When ${}^{d}x$ with $0 \leq d < D$ is the partition of x on device d the resulting formula is:

$${}^{d}x_i^{k+1} ={}^{d} D_i \cdot \left({}^{d}b_i - \sum_{j\neq i} {}^{d}a_{ij}x_j^k\right),$$

for $d \in \{0,\dots,D\}, i \in \{0,\dots,n/D\}$ and $j \in \{0,\dots,n-1\}$

When an iteration is completed each partition ${}^{d}x^{k+1}$ is copied to a temporary vector in RAM and distributed to each GPU. Basically the result of the parallelization is a matrix vector multiplication where a matrix ${}^{d}\mathbf{P} \in \mathbf{P}$ containing n/D rows of $\mathbf{P}$ and all columns is multiplied with x resulting in a vector ${}^{d}x'$ with n/D entries.

Algorithm 17.4: Host part of the Jacobi iteration for multiple GPUs

```
Input: $D$ number of GPUs, $x$ vector to multiply, $n$ number of entries in $x$,
Diag, $b$, rsize, non-zero, col matrix $\mathbf{P}$,
$\epsilon$ upper bound for distance, $\max_k$ upper bound for iterations
Output: $x'$ resulting vector, $k$ number of iterations
for $d \leftarrow 0 \dots D-1$ do in parallel                    {initialization loop}
    allocate memory for ${}^{d}x'$ ;
    allocate memory for ${}^{d}Diag$, ${}^{d}col$, ${}^{d}non-zero$, ${}^{d}b$, $x$, $\epsilon$, $n/D$ and copy ;
    allocate memory for TerminateGPU to be shared ;
    $Terminate_d \leftarrow$ false ;                          {array for early termination}
$rstart_0 \leftarrow 0$ ;                                    {conversion of the rsize array}
for $i \leftarrow 1 \dots |\text{rsize}|+1$ do                      {as described in Algorithm 17.2}
    $rstart_i \leftarrow rstart_{i-1} + rsize_{i-1}$;
allocate memory for rstartGPU, copy rstart to rstartGPU on each GPU ;
$k \leftarrow 0$ ;                                               {counter for iterations}
$blocks \leftarrow (n/D)/BlockSize + 1$ ;
{compute the number of blocks depending on the number of devices}
while (not $\bigwedge_{d=0}^{d<D}$ Terminate$_d$ and $k < \max_k$) do
{loop while at least one Terminate is set to false } and $\max_k$ not reached
    for $d \leftarrow 0 \dots D-1$ do in parallel                  {start all GPUs in parallel}
        copy x to VRAM ;
        <<<$blocks$,BlockSize>>>JacobiKernel()) ;
        copy TerminateGPU to $Terminate_d$ ;
        $beg \leftarrow d * (n/D)$ ;                        {compute first index for this device}
        $end \leftarrow (d+1) * (n/D)$ ;                     {compute last index for this device}
        copy ${}^{d}x'$ to $x_{beg,end}$ in RAM ;
    Swap(x,x');
    $k \leftarrow k+1$;
copy x' to RAM;
```

Algorithm 17.4 describes an approach to compute the resulting vector x^{k+1} by splitting the data of $\mathbf{P}$ and distributing it in the available memories. In (Bosnacki *et al.*, 2009) this approach is compared to an other more sophisticated approach and proved to be efficient on most of the evaluated instances. Compared to Algorithm 17.2 a number of modifications can be found. At the beginning, an array of *Terminate* variables containing an unique variable for each card, and instantiated in lines 9 and 10, has to be maintained, in contrast do a single variable. The main modification is a loop, executed in parallel on the host calling the kernel for each device. Since all kernels are started in parallel a synchronization point is necessary to ensure that the next kernel is not started before the results of all cards are available. The number of columns, and the size of x' in Algorithm 17.3 is determined from *rstartGPU* and used in the loop starting in line 8, hence the kernel can remain unchanged.

17.5 Summary

Having introduced the Jacobi iterations a probabilistic problem is mapped to, representations suitable for a GPU storage of matrices are proposed. Details on the implementation on a single GPU were given in section 3 of this chapter which closes with a proposal to utilize multiple GPUs. This part continues with an evaluation of the algorithms on a number of chosen protocols.

Chapter 18

Experimental evaluation

Three protocols, with different complexities were verified on each system. The following sections will describe the protocols and the achieved results in detail. The used hardware is described in Chapter 2

Structure of the chapter: Several protocols are evaluated to examine the efficiency of the proposed approach. Results are presented for the single and the multiple GPU implementation, and significant speedups achieved for all instances.

18.1 Verified Protocols

Three protocols, `herman`, `cluster` and `tandem`, shipped with the source of PRISM were evaluated. The protocols were chosen due to their scalability and the possibility to verify its properties by solving a linear function with the Jacobi method. Different protocols show different speedups achieved by the GPU, because the Jacobi iterations are only a part of the Model Checking algorithms, while the results show the time for the complete run.

The first protocol called `herman` is the Herman's self-stabilizing algorithm (Herman, 1990). The protocol operates synchronously on an oriented ring topology, i. e., the communication is unidirectional. The instance number denotes the number of processes in the ring, which must be odd. The underlying model is a DTMC. The verified PCTL property is 3 (`R=? [F "stable"{"k_tokens"}{max}]`) from the property file `herman.pctl`. Table 18.1 identifies this protocol as the one with the fewest lines and iterations, but also reveals the matrix of being of the largest density showing that the first instance uses about 1.34% of the cells, and the second instance 0.75%.

The second case study is `cluster` (Haverkort *et al.*, 2000) which models communication within a cluster of workstations. The system comprises two sub-clusters with N workstations (*instance* column in Table 18.1) in each of them, connected in a star topology. The switches connecting each sub-cluster are joined by a central backbone. All components can break down and there is a single repair unit to service all components. The underlying model is CTMC and the checked CSL property is property 1

Table 18.1: Detailed information on the protocol properties.

protocol	instance	n	iterations	GPU memory	non-zero cells
herman	15	32,768	245	55MB	1,342773 %
herman	17	131,072	308	495MB	0,755310 %
cluster	122	542,676	1,077	21 MB	0,001869 %
cluster	230	1,917,300	2,724	76 MB	0,000542 %
cluster	320	3,704,340	5,107	146 MB	0,000279 %
cluster	410	6,074,580	11,488	240 MB	0,000170 %
cluster	446	7,185,972	18,907	284 MB	0,000144 %
cluster	464	7,776,660	23,932	308 MB	0,000134 %
cluster	500	9,028,020	28,123	694 MB	0,000223 %
cluster	572	11,810,676	28,437	908 MB	0,000171 %
tandem	255	130,816	4,212	4 MB	0,006127 %
tandem	511	523,776	8,498	17 MB	0,001624 %
tandem	1,023	2,096,128	16,326	71 MB	0,000424 %
tandem	2,047	8,386,560	24,141	287 MB	0,000107 %
tandem	3,070	18,859,011	31,209	647 MB	0,000048 %
tandem	3,588	25,758,253	34,638	884 MB	0,000035 %
tandem	4,095	33,550,336	37,931	1,535 MB	0,000036 %

(`S=? [ "premium" ]`) from the corresponding property file. In this case study a sparser matrix was generated, which in turn needed more iterations to converge then the `herman` protocol. In the largest instance ($N = 572$) checked by the GPU, PRISM generates a matrix with 11,810,676 lines and iterates this matrix 28,437 times. It was even necessary to increase the maximum number of iterations, set by default to 10,000, to obtain a solution. Even though only one in 10,000 cells is used here the matrix uses up to 908 MB in the GPU, a fully filled matrix would consume over 500 terabyte of space, assuming each entry is stored in 4 bytes.

The third case study `tandem` is based on a tandem queuing network (Hermanns *et al.*, 1999). The model is represented as a CTMC which consists of a M/Cox(2)/1-queue sequentially composed with a M/M/1-queue. c is used to denote the capacity of the queues. Property 1 from the corresponding CSL property file (`R=? [ S ]`) is verified here. For this protocol Table 18.1 denotes the largest sparsity allowing a matrix with 25,758,253 lines to occupy 884 MB of graphics card memory resulting in a 0.000035% filling. Constant T was set to 1 for all experiments and parameter c was scaled as shown in the *instance* column of Table 18.1.

18.2 Empirical Results

In all tables of this section n denotes the number of rows (columns) of the matrix, "iterations" denotes the number of iterations of the Jacobi method, "CPU time" and "GPU Time" denote the runtimes of the standard (sequential) version of PRISM and our parallel implementation extension of the tool, respectively. All times are given in seconds. The speedup is computed as the quotient between the sequential and parallel runtimes.

All tables are partitioned into two parts, the first one showing the results for the 32-bit system, the second one for the 64-bit system. The first half does not contain the results for the multi-GPU implementation since a second GPU was not available in this system.

Table 18.2 shows the results of the verification using the implementation of the algorithm described in Section 17.3. Even though the number of iterations is rather small compared to the other models, the GPU achieves a speedup factor of approx. 1.5, and 0.9 on the 64-bit system. Since everything beyond multiplication of the matrix and vector is done on the CPU, the results prove the assumption that with a small matrix and a low number of iterations the memory transfer is to expensive compared to the benefit from the parallel computation. Unfortunately, it is not possible to scale up this model, due to the memory consumption being too high; the next possible instance (`herman19.pm`) consumes more then 2 GB. This table also reveals the differences between the used systems, while the clock time of the CPU differs only by about 30% the 64-bit system is more then twice as fast as the 32-bit one in the sequential mode. Here the GPU even slows down the verification process giving a factor of only 0.9. Due to the large density of the matrix adding a second GPU to the computation achieves a speedup in the larger instance, here the copying process, dominating the experiment, can be done in parallel to both GPUs, giving more time for the computation.

Table 18.2: Results for the `herman` protocol. (Times are given in seconds).

instance	CPU time	1 GPU	factor	2 GPUs	factor
15	22.430s	21.495s	1.04		
17	304.108s	206.174s	1.48		
15	10.544s	12.837s	0.82	12.135s	0.87
17	121.766s	140.248s	0.87	93.350s	1.30

Figure 18.1 shows that the GPU performs significantly better, Table 18.3 contains some exact numbers for chosen instances of the `cluster` protocol. The largest speedup reaches a factor of more then 9 on the 32-bit system and 6.6 on the other. Even for smaller instances, the GPU exceeds factor of 2. In this protocol, as well as in the next one, for large matrices a slight deterioration of the performance of the GPU implementation can be observed for which, for the time being, a clear explanation can not be given. One plausible hypothesis would be that after some threshold number of threads the GPU cannot profit any more from smart scheduling to hide the memory latencies. This experiment shows the costs of synchronizing the graphics cards by the host. The speedup converges to about 4 and slows down the computation compared to the usage of one GPU.

In the `tandem` protocol the best speedup was recorded shown in Table 18.4. For the best instance ($c = 2047$) PRISM generates a matrix with $8,386,560$ rows, which is iterated $24,141$ times. For this operation standard PRISM needs $9,672$ seconds while

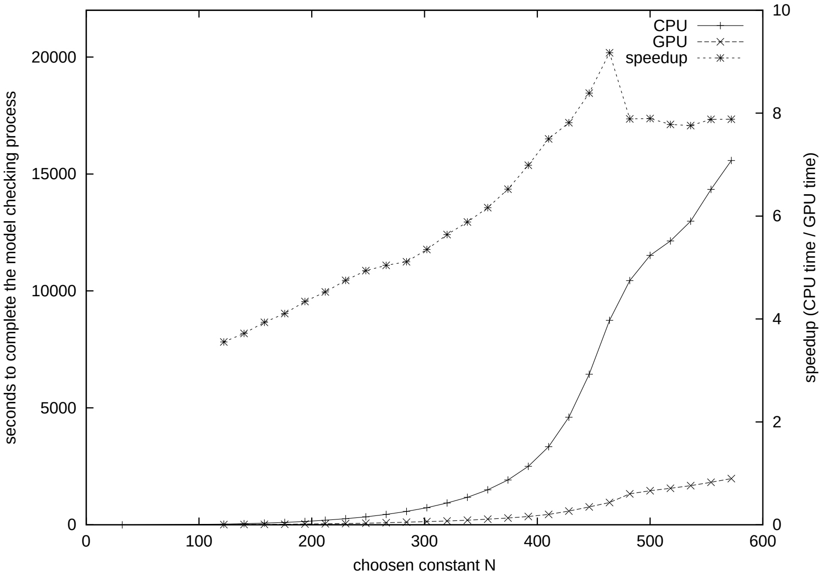


Figure 18.1: Verification times for several instances of the `cluster` protocol on the 32-bit system. The x-axis shows the value of the parameter N. Speedup is computed as described in the text as a quotient between the runtime of standard PRISM and the runtime of our GPU extension of the tool.

our parallel implementation only needs 516 seconds, scoring a maximal speedup of a factor 18.7 on the 32-bit system. Even on the faster 64-bit system the speedup is over one order of magnitude for all larger instances. Here the effect of synchronizing is even more obvious. Using both GPUs a speedup of a factor larger than four seems not achievable despite the fact that larger instances can be checked.

As mentioned above, 8 SPs share one double precision unit, but each SP has its own single precision unit. Hence, our hypothesis was that reducing the precision from double to single should bring a significant speedup. The code of PRISM was modified to support single precision floats for examining the effect. As can be seen in Figure 18.2 the hypothesis was wrong. The time per iteration in double precision mode is nearly the same as the single precision mode. The graph clearly shows that the GPU is able to hide the latency which occurs when a thread is waiting for the double precision unit by letting the SPs work on other threads. Nevertheless, it is important to note that the GPU with single precision arithmetic was able to verify larger instances of the protocol, given that the floating point numbers consumed less memory.

It should be noted that in all case studies the MTBDD and hybrid representations of the models, which are an option in PRISM, were studied, but in all cases the running times were consistently slower than the ones with the sparse matrix representation, which are shown in the tables.

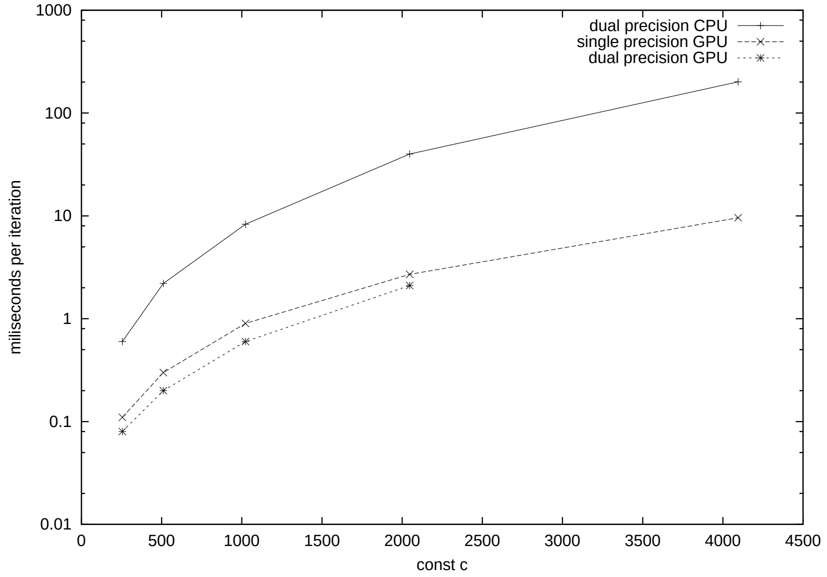


Figure 18.2: Time per iteration on the 32-bit system in the `tandem` protocol. The CPU is significantly slower then the GPU operating in single or double precision. Reducing the precision has nearly no effect on the speed.

18.3 Summary

This part introduced GPU probabilistic/stochastic Model Checking as a novel concept. To this end it described a parallel version of Jacobi's method for solving linear equations, which is the main core of the algorithms for Model Checking discrete- and continuous-time Markov chains, i. e., the corresponding logics PCTL and CSL. The algorithm was implemented on top of the probabilistic model checker PRISM. The efficiency and the advantages of the GPU Probabilistic Model Checking in general were illustrated on several case studies. Speedups of up to 18 times compared to the sequential implementation of PRISM were achieved. On a recent system the speedup still reaches a factor of 15.

Table 18.3: Results for the `cluster` protocol. Parameter N is used to scale the protocol.

N	CPU time	1 GPU	factor	2 GPUs	factor
122	31.469s	8.855s	3.55	.	
230	260.440s	54.817s	4.75	.	
320	931.515s	165.179s	5.63	.	
410	3,339.307s	445.297s	7.49	.	
446	6,440.959s	767.835s	8.38	.	
464	8,739.750s	952.817s	9.17	.	
500	11,516.716s	1,458.609s	7.89	.	
572	15,576.977s	1,976.576s	7.88	.	
122	16.400s	6.906s	2.37	8.620s	1.90
230	135.269s	34.732s	3.89	46.685s	2.90
320	469.827s	101.664s	4.62	141.456s	3.32
410	1,649.663s	286.014s	5.77	429.626s	3.84
446	3,143.487s	512.708s	6.13	785.629s	4.00
464	4,270.262s	643.850s	6.63	1,024.335s	4.17
500	4,865.687s	1,027.095s	4.73	1,470.256s	3.30
572	6,630.097s	1,386.101s	4.78	1,964.418s	3.38

Table 18.4: Results from the verification of the `tandem` protocol. The constant c is used to scale the protocol (o.o.m. denotes out of global memory).

c	CPU time	1 GPU	factor	2 GPUs	factor
255	26.99s	3.63s	7.4	.	
511	190.26s	17.80s	10.7	.	
1,023	1,360.58s	103.15s	13.2	.	
2,047	9,672.19s	516.33s	18.7	.	
3,070	25,960.39s	1,502.85s	17.3	.	
3,588	33,820.21s	2,435.41s	13.9	.	
4,095	76,311.59s	o.o.m.		.	
255	14.96s	3.56s	4.20	6.79s	2.20
511	98.81s	11.42s	8.65	27.89s	3.54
1,023	658.78s	65.51s	10.06	166.27s	3.97
2,047	3,642.62s	384.68s	9.47	946.76s	3.85
3,070	10,049.93s	866.27s	11.60	2,510.67s	4.00
3,588	15,114.93s	1,319.13s	11.46	3,794.79s	3.98
4,095	22,174.01s	o.o.m.		5,386.44s	4.12

Part VI

Conclusions and Future Work

Chapter 19

Conclusion

Each part summarized results of itself, only, therefore a comprehensive overview will be given in the reminder of this work. The achieved results in each part will be summarized and a discussion on future developments given in the reminder.

19.1 Conclusions

This work started with an introduction to state spaces, defining the prerequisites needed to understand the process of searching in an implicit graph. Definition 2, introducing the *state* is seen as the center of this work. It defines the most prominent element which glues together all the algorithms introduced mainly to expand and handle states. Subsequently, an example was introduced describing the process of a student writing a thesis. Although this example is far to be a realistic representation of the problem, it suffices to give an insight on the challenges dealt in this work.

Next, graph search algorithms were presented, starting with the pseudo code of an algorithm utilizing just one structure to store states, i. e., $Open$. The algorithms end with the introduction of a naive parallel approach, being the starting point for this work. The main problem with states in state spaces is the huge number of them. Not only expansion may take a serious amount of time, they need to be stored, also. Thus, Section 1.4 introduced known strategies for an efficient duplicate detection on external storage, for a nearly unlimited size, and in internal memory, for a short response time.

This work is on utilizing novel hardware to accelerate the state space search. Utilizing the hardware requests a knowledge on the intern structures given in Chapter 2. Here the differences of solid state media and magnetic media were sketched. While SSDs store information electronically on erasable memory chips, HDDs use a spinning disk to store data. The main difference, evaluated in the scope of this work is the short random access latency of solid state media. While the hard drive needs about 7ms to move the read/write head to the appropriate position, an SSD delivers the data a 100 times faster.

The increased storage capability and access speed enabling the examination of larger state spaces provided, enough time for the generation is given. To decrease the searching time a parallel expansion of states is useful, e. g., on the highly parallel processor of a graphics card, introduced and evaluated also in the second chapter.

GPUSSD-Breadth-First search The contribution starts with giving the prerequisites for an efficient GPU and SSD utilization in state space searching. The searching algorithm is examined for tasks suitable to be ported to the GPU, followed by a discussion where to store the generated states. Having examined the searching process a framework on using GPUs and SSDs in searching was proposed. This framework divides the process into three stages namely, evaluation of the states for existing successors, generation of the successors and removal of expanded states in a duplicate detection.

For each stage a strategy was presented on how to perform the task efficiently on the GPU or to use SSD storage in duplicate detection. The presented strategies were divided into a successor counting strategy and a successor pointing strategy. The first one counts the successors available in a state. The number is used to allocate enough global memory on the graphics card for the generation. When a state is expanded the preconditions have to be checked again since only the number of successors is known in this strategy. The second strategy, the successor pointing strategy classifies the transitions of a state in groups of active and non-active transitions. It makes an reevaluation of the preconditions in the generation step needles but requires more memory. The chapter continued with the proposal to convert the pre- and postconditions to the Reverse Polish Notation due to its pointer less flat representation suitable for the GPU. It closed with an external GPU searching approach utilizing RAM only for buffers needed to transfer data from disk to the GPU.

Explicit State Model Checking In the second part the proposed approach was applied to explicit state Model Checking externalizing the verification to solid state media and increasing the searching speed by using graphics cards for the generation of the state space. Having introduced Explicit State Model Checking and given an overview of the related work, a semi-external algorithm was proposed. This algorithm utilizes perfect hashing to store a portion of the information, necessary to traverse a graph, on solid state media. Based on the internal minimal counterexamples algorithm from Gastin and Moro (2007) an additional BFS was inserted at the beginning to generate a perfect hash function. The algorithm performs three stages to generate the minimal counterexample, the generation of the perfect hash function using an external BFS, the search for the counterexample using internal memory and an SSD, and the reconstruction of the counterexample. The perfect hash function is generated and stored efficiently on SSDs and profits from the increased random access time of this media while searching. Having externalized it each BFS traversal can be done using internal memory filled nearly completely with a 1-bit $Closed$ list.

The minimal counter example search time is dominated by the generation of the whole state space for the perfect hash function. This work is done no matter if a counter example exists or not, so a fast generation method is mandatory.

Since the evaluation of a guard and the generation of a successor using the effects can be separated in Explicit State Model Checking the successor pointing strategy was was applied here. The generation was divided in three stages according to the settings given by the framework. Duplicate detection was performed either as a sorting based strategy, including the not complete state compression method and by utilizing a parallel bloom filter. To perform the evaluation of a guard and to evaluate the effects on the GPU a flat representation is needed which can be evaluated by a sequential traversal. Here the chosen polish Reverse Polish Notation has shown good results in terms of model description size and searching time.

The evaluation has revealed the approach to be efficient in utilizing additional memory for duplicate detection and parallelizing the search on the highly parallel graphics processing unit. Both algorithms show promising results in either increasing the size of the expanded state space or the acceleration of the searching process.

Action Planning In this part Action Planning was successfully enhanced by the usage of the graphics card to speed up the search and external media to increase storage space by outsourcing $Open$. The first domain-independent planner CUDPLAN is proposed that exploits the power available on the graphics card. The flat representation of the preconditions and effects was realized by converting them to a postfix notation. While $Open$ is outsourced to external devices connected in a RAID to increase transferring speed, $Closed$ remains in the internal memory due to the high random access speed. Parallelization of the duplicate detection is performed by utilizing a lock-less hashing approach and storing states in a hash table. Due to the state sizes a sorting based approach on the GPU is not effective and the optimality criteria prohibits the usage of compression strategies which forgo information.

The searching process was divided into three stages and the successor pointing strategy used. Lowering the number of expanded states is achieved by using a Dijkstra search and partitioning $Open$ by means of the path costs.

The approach has shown to be effective in finding optimal plans and achieves a decent speedup compared to sequential implementations.

Game Solving Having described the analyzed games and given hints on their state spaces the chapter sketches the strategies in game solving. In one person combinatorial games a forward search suffices to determine whether the game is won. However, in most games a backward search is necessary to propagate the game theoretical results to the initial state. In all games compression methods to store the states come in handy due to the large number of states. Perfect Hash strategies were presented in Chapter 13 and divided in three groups depending on the game to hash. Using this functions a two-bit BFS traversal of the state space can be performed or even an one-bit reachability analysis. Such algorithms, proposed and ported to the GPU in Chapter 14, were evaluated in the remainder of the Game Solving part. The evaluation has shown significant speedups of a factor about 27 compared to a single thread CPU implementation. A case study performed here was the solution to the game Nine-Men-Morris which is a two player game with indistinguishable pieces. The work validates Gasser's results revealing the game to be a draw.

Probabilistic Model Checking In the domain of Probabilistic Model Checking a state space exploration is possible but inefficient. Here numerical approaches show better results and a different strategy for the GPU has to be considered. The work proposed to port the Jacobi iterations, the most time demanding part of the checking process, to the GPU. Part V describes Probabilistic Model Checking and introduces the checking process. In the second chapter of this part the matrix vector multiplication is ported to the GPU and an efficient matrix representation presented. The algorithm was extended to use multiply GPUs, to increase the size of possible state spaces, and evaluated. It shows significant speedups of about one order of magnitude compared to the original sequential approach.

19.2 Future Work

It is always hard to predict the future, and especially in a topic like hardware developments. Although the impact of SSDs in computing is rising, the specifications differ with new generations. Recent developments in random writing speed of such devices show a significant increase of the throughput, so a possible utilization of the hash based GPU sorting approach on external memory could be evaluated. This work also revealed a RAID of SSDs being faster then a single disk putting hash based strategies into the focus of external duplicate detection. The question whether SSD storage should be used in place of RAM is a question of costs and it will remain one. Both memories are chip devices bound to Moore's Law so using RAM will remain the expensive solution since the capacity per chip is predicted to double every two years at the same price.

The development in graphics hardware is an ongoing process, also. The manufacturers are concatenating more and more parallel processors to have more computing power available. Additionally, when this work started nearly no interest in GPU utilization in science existed, today many publications on this topic are available. In NVIDIA's latest Fermi architecture the separation of the streaming multiprocessors in texture processing clusters was eliminated decreasing the distance to the global memory. Additionally the shared memory and the registers were extended. The concentration on memory in the structure follows the demands of using GPUs in general purpose programming, where an efficient storage of information is mandatory. New algorithms should be developed to profit from this developments.

In contrast to following the developments in hardware the disciplines are by far not examined to a full extend in this work. CUDMOC is just able to verify safety properties due to the utilization of BFS and will be extended to check lifeness properties in the feature.

One obvious way to go from here is *heuristic search* using algorithms like A* and porting them to the GPU. Evaluations has shown that it is not the processing power of the GPU which limits the speed-ups but the memory access speed. Additional test computations while expanding states in Action Planning have shown no impact on the searching speed. This shows that a computation of a heuristic value would come for free in terms of time.

The presented approaches are not bind to specific GPU types, not even to graphic cards. When describing the aspects of a system equipped with a GPU we speak about an infinite external storage and a finite amount of RAM connected over a rather slow connection to a large number of processing cores. This image can be ported nearly unchanged to a cluster of computers with a centralized storage. Here, the GPUs are mapped to the unique cores, and the slow connection is the network. However this mapping is not completely adequate since the GPU cores are connected to a global VRAM while the unique cluster nodes are not. Having realized the similarity's of both systems porting algorithms to cluster computing modified according to the framework is possible. It will be interesting to evaluate the presented framework and different strategies on other systems then one with one ore more GPUs.

Finally, a fine grained theoretical model of the GPU devices would be very helpful in the development and analysis of GPU algorithms.

Bibliography

James Abello, Adam L. Buchsbaum, and Jeffery Westbrook. A functional approach to external graph algorithms. In Gianfranco Bilardi, Giuseppe F. Italiano, Andrea Pietracaprina, and Geppino Pucci, editors, *Proceedings of the 6th Annual European Symposium on Algorithms (ESA)*, Lecture Notes in Computer Science, pages 332–343, London, UK, 1998. Springer.

Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communication of the ACM*, 31(9):1116–1127, 1988.

Sudhir Aggarwal, R. Alonso, and Costas Courcoubetis. Distributed reachability analysis for protocol verification environments. In *Discrete Event Systems: Models and Application*, volume 103 of *Lecture Notes in Control and Information Sciences*, pages 40–56. Springer, 1987.

Deepak Ajwani, Ulrich Meyer, and Vitaly Osipov. Improved external memory BFS implementations. In *Proceedings of the Algorithm Engineering and Experiments Workshop (ALENEX)*, pages 3–12, 2007.

Deepak Ajwani, Itay Malinger, Ulrich Meyer, and Sivan Toledo. Graph search on flash memory, 2008. MPI-TR.

Deepak Ajwani, Andreas Beckmann, Riko Jacob, Ulrich Meyer, and Gabriel Moruz. On computational models for flash memory devices. In Jan Vahrenhold, editor, *Experimental Algorithms*, volume 5526 of *Lecture Notes in Computer Science*, pages 16–27. Springer, 2009.

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

Christel Baier, Boudewijn Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model-checking algorithms for continuous-time markov chains. *IEEE Transactions on Software Engineering*, 29(6):524–541, 2003.

Walter William Rouse Ball. *Mathematical recreations and essays*. Macmillan and Co., limited, 1911.

Jiri Barnat, Lubos Brim, and Jakub Chaloupka. From distributed memory cycle detection to parallel model checking. *Electronic Notes in Theoretical Computer Science*, 133:21–39, 2005.

Jiri Barnat, Lubos Brim, and Pavel Šimeček. I/O efficient accepting cycle detection. In Holger Hermanns Werner Damm, editor, *Proceedings of the 19th International Conference on Computer Aided Verification(CAV)*, Lecture Notes in Computer Science, pages 281–293. Springer, 2007.

Jiri Barnat, Lubos Brim, Stefan Edelkamp, Pavel Šimeček, and Damian Sulewski. Can flash memory help in model checking? In Darren D. Cofer and Alessandro Fantechi, editors, *Formal Methods for Industrial Critical Systems, 13th International Workshop (FMICS)*, Lecture Notes in Computer Science, pages 150–165. Springer, 2008.

Jiri Barnat, Lubos Brim, Pavel Šimeček, and Michael Weber. Revisiting resistance speeds up I/O-efficient LTL model checking. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science, pages 48–62. Springer, 2008.

Jiri Barnat, Lubos Brim, Milan Ceska, and Tomas Lamr. CUDA accelerated LTL Model Checking. In *Proceedings of the 15th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 34–41. IEEE Computer Society, 2009.

Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceska. Employing multiple cuda devices to accelerate ltl model checking. In *IEEE 16th International Conference on Parallel and Distributed Systems, ICPADS 2010, 8-10 Dec. 2010, Shanghai, China*, pages 259–266. IEEE, 2010.

Jiri Barnat, Lubos Brim, and Petr Rockai. Scalable shared memory ltl model checking. *STTT*, 12(2):139–153, 2010.

Jiri Barnat, Petr Bauch, Lubos Brim, and Milan Ceska. Computing strongly connected components in parallel on cuda. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*, pages 544–555. IEEE, 2011.

Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference (AFIPS)*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, 1968.

Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In Amos Fiat and Peter Sanders, editors, *Proceedings of the 17th Annual European Symposium on Algorithms (ESA)*, volume 5757 of *Lecture Notes in Computer Science*, pages 682–693. Springer, 2009.

Alexander Bell and Boudewijn R. Haverkort. Distributed disk-based algorithms for model checking very large markov chains. *Formal Methods in System Design*, 29:177–196, 2006.

Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for your Mathematical Plays*, volume 2. Academic Press, ISBN 0-12-091152-3, 1982. chapter 25.

Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *Lecture Notes in Computer Science*. Springer, 1999.

Burton H. Bloom. Space/time trade-offs in hashing coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

David M. Bloom. A birthday problem. *American Mathematical Monthly*, 80:1141–1142, 1973.

Blai Bonet. Efficient algorithms to rank and unrank permutations in lexicographic order. In *AAAI-Workshop on Search in AI and Robotics*, 2008.

Dragan Bosnacki, Stefan Edelkamp, and Damian Sulewski. Efficient probabilistic model checking on general purpose graphics processors. In *Model Checking Software, 16th International SPIN Workshop*, volume 5578 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2009.

Dragan Bosnacki, Stefan Edelkamp, Damian Sulewski, and Anton Wijs. Parallel probabilistic model checking on general purpose graphics processors. *STTT*, 13(1):21–35, 2011.

Fabiano C. Botelho and Nivio Ziviani. External perfect hashing for very large key sets. In Mário J. Silva, Alberto H. F. Laender, Ricardo A. Baeza-Yates, Deborah L. McGuinness, Bjørn Olstad, Øystein Haug Olsen, and André O. Falcão, editors, *Proceedings of the 16th ACM Conference on Information and Knowledge Management (CIKM)*, pages 653–662. AAAI Press, 2007.

Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Norbert Zeh, editors, *Algorithms and Data Structures, 10th International Workshop*, volume 4619 of *Lecture Notes in Computer Science*, pages 139–150. Springer, 2007.

Gilles Brassard and Paul Bratley. *Fundamentals of algorithms*. Prentice Hall, 1996.

Lubos Brim, Ivana Černá, Pavel Moravec, and Jirí Simsa. Accepting predecessors are better than back edges in distributed LTL model-checking. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, volume 3312 of *Lecture Notes in Computer Science*, pages 352–366, 2004.

Ilja N. Bronshtein and Konstantin A. Semendyayev. *Handbook of mathematics (3rd ed.)*. Springer, London, UK, 1997.

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics (TOG)*, 23:777–786, 2004.

Olaf Burkart and Bernhard Steffen. Model checking the full modal mu-calculus for infinite sequential processes. In *International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1256 of *Lecture Notes in Computer Science*, pages 419–429. Springer, 1997.

Arthur W. Burks, Don W. Warren, and Jesee B. Wright. An analysis of a logical machine using parenthesis-free notation. *Mathematical Tables and Other Aids to Computation*, 8(46):53–57, 1954.

Ethan Burns, Seth Lemons, Wheeler Ruml, and Rong Zhou. Suboptimal and anytime heuristic search on multi-core machines. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 42–49, 2009.

Ethan Burns, Seth Lemons, Rong Zhou, and Wheeler Ruml. Best-first heuristic search for multi-core machines. In Craig Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 449–455, 2009.

Daniel Cederman and Philippas Tsigas. A practical quicksort algorithm for graphics processors. In Dan Halperin and Kurt Mehlhorn, editors, *Proceedings of the 16th Annual European Symposium on Algorithms (ESA)*, Lecture Notes in Computer Science, pages 246–258. Springer, 2008.

Ting Chen and Steven Skiena. Sorting with fixed-length reversals. *Discrete Applied Mathematics*, 71(1–3):269–295, 1996.

Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

Gene Cooperman and Larry Finkelstein. New methods for using Cayley graphs in interconnection networks. *Discrete Applied Mathematics*, 37/38:95–118, 1992.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill, 2001.

Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2-3):275–288, 1992.

Roman Dementiev, Lutz Kettner, and Peter Sanders. STXXL: Standard template library for XXL data sets. In Gerth Stølting Brodal and Stefano Leonardi, editors, *Proceedings of the 13th Annual European Symposium on Algorithms (ESA)*, Lecture Notes in Computer Science, pages 640–651. Springer, 2005.

Robert B. Dial. Shortest-path forest with topological ordering. *Communication of the ACM*, 12(11):632–633, 1969.

Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal of Computing*, 23:738–761, 1994.

Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

Harry Dweighter. Problem e2569. *American Mathematical Monthly*, (82):1010, 1975.

Stefan Edelkamp and Shahid Jabbar. Cost-optimal external planning. In *Proceedings of the 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference (AAAI)*. AAAI Press, 2006.

Stefan Edelkamp and Shahid Jabbar. Large-scale directed model checking LTL. In *Model Checking Software, 13th International SPIN Workshop*, volume 3925 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2006.

Stefan Edelkamp and Shahid Jabbar. MIPS-XXL: Featuring external shortest path search for sequential optimal plans and external branch-and-bound for optimal net benefit. In *Proceedings of the 6th International Planning Competition (IPPC)*, 2008.

Stefan Edelkamp and Peter Kissmann. Symbolic exploration for general game playing in PDDL. In *ICAPS-Workshop on Planning in Games*, 2007.

Stefan Edelkamp and Peter Kissmann. Optimal symbolic planning with action costs and preferences. In Craig Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1690–1695, 2009.

Stefan Edelkamp and Alberto Lluch-Lafuente. Abstraction in directed model checking. In *ICAPS-Workshop on Connecting Planning Theory with Practice*, 2004.

Stefan Edelkamp and Stefan Schroedl. *Heuristic Search: Theory and Practice.* Morgan Kaufmann, 2011.

Stefan Edelkamp and Damian Sulewski. Flash-efficient LTL model checking with minimal counterexamples. Technical Report 820, Department of Computer Science, Dortmund University of Technology, Germany, 2008.

Stefan Edelkamp and Damian Sulewski. Flash-Efficient LTL Model Checking with Minimal Counterexamples. In Antonio Cerone and Stefan Gruner, editors, *Proceedings of the 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, pages 73–82. IEEE Computer Society, 2008.

Stefan Edelkamp and Damian Sulewski. Model checking via delayed duplicate detection on the GPU. Technical Report 821, Technische Universität Dortmund, 2008. Presented on the 22nd Workshop on Planning, Scheduling, and Design PUK 2008.

Stefan Edelkamp and Damian Sulewski. Parallel state space search on the GPU. Electronicaly, 2009.

Stefan Edelkamp and Damian Sulewski. Efficient explicit-state model checking on general purpose graphics processors. In Jaco van de Pol and Michael Weber, editors, *Model Checking Software - 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010. Proceedings*, volume 6349 of *Lecture Notes in Computer Science*, pages 106–123. Springer, 2010.

Stefan Edelkamp and Damian Sulewski. External memory breath-first search with delayed duplicate detection on the GPU. In Ron van der Meyden and Jan-Georg Smaus, editors, *Proceedings of the 6th Workshop on Model Checking and Artificial Intelligence (MoChArt-2010)*, Atlanta, GA, 2010. Springer.

Stefan Edelkamp and Ingo Wegener. On the performance of weak-heapsort. In *Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '00, pages 254–266, London, UK, 2000. Springer.

Stefan Edelkamp, Shahid Jabbar, and Stefan Schrödl. External A**. In Susanne Biundo, Thom W. Frühwirth, and Günther Palm, editors, *Proceedings of the 27th Annual German Conference on AI Advances in Artificial Intelligence (KI)*, volume 3238 of *Lecture Notes in Computer Science*, pages 226–240. Springer, 2004.

Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology*, 5:247–267, 2004.

Stefan Edelkamp, Shahid Jabbar, and Mohammed Nazih. Cost-optimal planning with constraints and preferences in large state spaces. In *ICAPS-Workshop on Preferences and Soft Constraints in Planning*, pages 38–45, 2006.

Stefan Edelkamp, Shahid Jabbar, and Damian Sulewski. Distributed verification of multi-threaded C++ programs. *Electronic Notes in Theoretical Computer Science*, 198(1):33–46, 2008.

Stefan Edelkamp, Peter Sanders, and Pavel Šimeček. Semi-external LTL model checking. In Sharad Malik Aarti Gupta, editor, *Proceedings of the 20th International Conference on Computer Aided Verification(CAV)*, volume 5123 of *Lecture Notes in Computer Science*, pages 530–542. Springer, 2008.

Stefan Edelkamp, Mark Kellershoff, and Damian Sulewski. Program model checking via action planning. In Ron van der Meyden and Jan-Georg Smaus, editors, *Model Checking and Artificial Intelligence - 6th International Workshop, MoChArt 2010, Atlanta, GA, USA, July 11, 2010, Revised Selected and Invited Papers*, volume 6572 of *Lecture Notes in Computer Science*, pages 32–51. Springer, 2010.

Stefan Edelkamp, Damian Sulewski, and Cengizhan Yücel. GPU exploration of two-player games with perfect hash functions. In Ariel Felner and Nathan R. Sturtevant, editors, *Proceedings of the Third Annual Symposium on Combinatorial Search, SOCS 2010, Stone Mountain, Atlanta, Georgia, USA, July 8-10, 2010*. AAAI Press, 2010.

Stefan Edelkamp, Damian Sulewski, and Cengizhan Yücel. Perfect hashing for state space exploration on the GPU. In Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz, editors, *ICAPS*, pages 57–64. AAAI Press, 2010.

Stefan Edelkamp, Damian Sulewski, Jiri Barnat, Lubos Brim, and Pavel Simecek. Flash memory efficient LTL model checking. *Science of Computer Programming*, 76(2):136–157, 2011.

Ulf Eickmann. Untersuchung der Echtzeitfähigkeit von Budget-Grafikkarten. Diplomarbeit, The German Film School Elstal, 2004.

Markus Enzenberger and Martin Müller. A lock-free multithreaded Monte-Carlo tree search algorithm. In *Advances in Computer Games*, pages 14–20, 2009.

Matthew P. Evett, James A. Hendler, Ambuj Mahanti, and Dana S. Nau. PRA*: Massively parallel heuristic search. *Journal on Parallel and Distributed Computation*, 25(2):133–143, 1995.

Jean-Claude Fernandez, Laurent Mounier, Claude Jard, and Thierry Jéron. On-the-fly verification of finite transition systems. *Formal Methods in System Design*, 1:251–273, 1992.

Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

Ralph Gasser. Solving nine men's morris. *Computational Intelligence*, 12:24–41, 1996.

Paul Gastin and Pierre Moro. Minimal counterexample generation in SPIN. In *Model Checking Software, 14th International SPIN Workshop*, volume 4595 of *Lecture Notes in Computer Science*, pages 24–38. Springer, 2007.

Willialm H. Gates and Christos H. Papadimitriou. Bounds for sorting by prefix reversal. *Discrete Mathematics*, 27:47–57, 1979.

Ratan K. Ghosh and G. P. Bhattacharjee. A parallel search algorithm for directed acyclic graphs. *BIT*, 24(2):134–150, 1984.

Ratan K. Ghosh. Parallel search algorithms for graphs and trees. *Information Sciences*, 67(1-2):137 – 165, 1993.

Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Lecture Notes in Computer Science. Springer, 1996.

Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High performance graphics coprocessor sorting for large database management. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 325–336. ACM Press, 2006.

Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:102–111, 1994.

Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using cuda. In Srinivas Aluru, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna, editors, *High Performance Computing (HiPC)*, Lecture Notes in Computer Science, pages 197–208. Springer, 2007.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Science and Cybernetics*, 4(2):100–107, 1968.

Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. On the use of model checking techniques for dependability evaluation. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 228–237, Erlangen, Germany, 2000. IEEE Computer Society.

Ted Richard Herman. Probabilistic self-stabilization. *Information Processing Letters*, 35(2):63–67, 1990.

Holger Hermanns, Joachim Meyer-Kayser, and Markus Siegle. Multi terminal binary decision diagrams to represent and analyse continuous time Markov chains. In B. Plateau, W. Stewart, and M. Silva, editors, *Proceedings of the 3rd International Workshop on Numerical Solution of Markov Chains (NSMC'99)*, pages 188–207. Prensas Universitarias de Zaragoza, 1999.

Charles Antony Richard Hoare. Communicating sequential processes. *Communication of the ACM*, 21(8):666–677, 1978.

Douglas R. Hofstadter. *Gdel, Escher, Bach.* Basic Books, 1979.

Gerard Holzmann and Dragan Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, 2007.

Gerard J. Holzmann and Anuj Puri. A minimized automaton representation of reachable states. *International Journal on Software Tools for Technology Transfer*, 2(3):270–278, 1999.

Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 1–6. IEEE, 2008.

Gerard J. Holzmann. An analysis of Bitstate hashing. *Formal Methods in System Design*, 13(3):287–305, 1998.

Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2004.

Edward Hordern. *Sliding Piece Puzzles.* Oxford University Press, 1986.

Shahid Jabbar and Stefan Edelkamp. I/O efficient directed model checking. In Radhia Cousot, editor, *Proceedings of the 7th International Conference Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *Lecture Notes in Computer Science*, pages 313–329. Springer, 2005.

Shahid Jabbar and Stefan Edelkamp. Parallel external directed model checking with linear I/O. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Proceedings of the 6th International Conference Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3855 of *Lecture Notes in Computer Science*, pages 237–251. Springer, 2006.

Shahid Jabbar. *External Memory Algorithms for State Space Search in Model Checking and Action Planning*. PhD thesis, Dortmund University of Technology, 2008.

Hajek Jan. Progress report on the automatic and proven protocol verifier. *Computer Communication Review ACM SigComm*, 8(1):15–16, 1978.

Harold Jeffreys and Bertha S. Jeffreys. *Methods of Mathematical Physics, 3rd ed.* Cambridge University Press, 1988.

Kinam Kim, Jung Hyuk Choi, Jungdal Choi, and Hong-Sik Jeong. The future prospect of nonvolatile memory. *VLSI Technology*, pages 88–94, 2005.

Akihiro Kishimoto, Alexander S. Fukunaga, and Adi Botea. Scalable, parallel best-first search for optimal sequential planning. In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 201–208. AAAI Press, 2009.

Peter Kissmann and Stefan Edelkamp. Solving fully-observable non-deterministic planning problems via translation into a general game. In Bärbel Mertsching, Marcus Hund, and Muhammad Zaheer Aziz, editors, *Proceedings of the 32nd Annual German Conference on AI Advances in Artificial Intelligence (KI)*, volume 5803 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 2009.

Donald E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison Wesley, 1973.

Richard E. Korf and Peter Schultze. Large-scale parallel breadth-first search. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings of the 20th National Conference on Artificial Intelligence and the 17th Innovative Applications of Artificial Intelligence Conference (AAAI)*, pages 1380–1385. AAAI Press / The MIT Press, 2005.

Richard E. Korf and Weixiong Zhang. Divide-and-conquer frontier search applied to optimal sequence alignment. In *Proceedings of the 17th National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence (AAAI)*, pages 910–916. AAAI Press / The MIT Press, 2000.

Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

Richard E. Korf. Divide-and-conquer bidirectional search: First results. In Thomas Dean, editor, *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1184–1189. Morgan Kaufmann, 1999.

Richard E. Korf. Breadth-first frontier search with delayed duplicate detection. In *Model Checking and Artificial Intelligence (MoChArt)*, pages 87–92, 2003.

Richard E. Korf. Delayed duplicate detection: extended abstract. In Georg Gottlob and Toby Walsh, editors, *IJCAI'03: Proceedings of the 18th international joint Conference on Artificial Intelligence*, pages 1539–1541, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

Richard E. Korf. Frontier search. *Journal of the ACM*, 52(5):715–748, 2005.

Richard E. Korf. Linear-time disk-based implicit graph search. *Journal of the ACM*, 55:26:1–26:40, 2008.

Richard E. Korf. Minimizing disk I/O in two-bit-breath-first search. In Carla P. Gomes Dieter Fox, editor, *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI)*, pages 317–324. AAAI Press, 2008.

Daniel Kunkle and Gene Cooperman. Twenty-six moves suffice for Rubik's cube. In Dongming Wang, editor, *ISSAC'07: Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, pages 235–242, New York, NY, USA, 2007. ACM.

Marta Kwiatkowska, Gethin Norman, and David Parker. Probabilistic symbolic model checking with prism: a hybrid approach. *Int. J. Softw. Tools Technol. Transf.*, 6:128–142, August 2004.

Marta Kwiatkowska, Gethin Norman, and David Parker. Stochastic model checking. In Marco Bernardo and Jane Hillston, editors, *Proceedings of the 7th international conference on Formal methods for performance evaluation (SFM)*, volume 4486 of *Lecture Notes In Computer Science*, pages 220–270, Berlin, Heidelberg, 2007. Springer.

Alfons Laarman, Jaco Pol van de, and Michael Weber. Boosting multi-core reachability performance with shared hash tables. In N. Sharygina and R. Bloem, editors, *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. IEEE Computer Society, 2010.

Alfons Laarman, Jaco van de Pol, and Michael Weber. Multi-core ltsmin: Marrying modularity and scalability. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, pages 506–511. Springer, 2011.

Alfons W. Laarman, Rom Langerak, Jaco van de Pol, Michael Weber, and Anton Wijs. Multi-core nested depth-first search. In T. Bultan and P-A. Hsiung, editors, *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis, ATVA 2011, Tapei, Taiwan*, volume online pre-publication of *Lecture Notes in Computer Science*, London, July 2011. Springer.

Peter Lamborn and Eric Hansen. Layered duplicate detection in external-memory model checking. In *Model Checking Software, 15th International SPIN Workshop*, volume 5156 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2008.

Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. GPU sample sort. In *Proceedings of the 24th International Symposium on Parallel and Distributed Processing, (IPDPS)*, pages 1–10, 2010.

Ryan Lichtenwalter and Nitesh V. Chawla. Disnet: A framework for distributed graph computation. In *International Conference on Advances in Social Networks Analysis and Mining, ASONAM 2011, Kaohsiung, Taiwan, 25-27 July 2011*, pages 263–270. IEEE Computer Society, 2011.

Alberto Lluch-Lafuente, Stefan Leue, and Stefan Edelkamp. Partial order reduction in directed model checking. In *Model Checking Software, 9th International SPIN Workshop*, pages 112–127, 2002.

Drew McDermott. PDDL – the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993.

Kurt Mehlhorn and Ulrich Meyer. External-memory breadth-first search with sublinear I/O. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA)*, volume 2461 of *Lecture Notes in Computer Science*, pages 723–735. Springer, 2002.

Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn, editors. *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*, volume 2625 of *Lecture Notes in Computer Science*. Springer, 2003.

Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

Gordon E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8):114–117, 1965.

Markus Müller-Olm, David Schmidt, and Bernhard Steffen. Model-Checking: A Tutorial Introduction. In *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 330–354. Springer, 1999.

Kamesh Munagala and Abhiram G. Ranade. I/O-complexity of graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, pages 687–694, 1999.

Wendy J. Myrvold and Frank Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79(6):281–284, 2001.

Hootan Nakhost, Jörg Hoffmann, and Martin Mueller. Improving local search for resource-constrained planning. In *Symposium on Combinatorial Search (SoCS)*, pages 81–82, 2010.

Kedar S. Namjoshi and Robert P. Kurshan. Syntactic program transformations for automatic abstraction. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 435–449. Springer, 2000.

Olavi Nevanlinna. *Convergence of iterations for linear equations.* Lectures in Mathematics ETH Zürich. Springer, 1993.

Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In Friedhelm Meyer auf der Heide, editor, *Proceedings of the 9th Annual European Symposium on Algorithms (ESA)*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2001.

Judea Pearl. *Heuristics*. Addison-Wesley, 1985.

Radek. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *Model Checking Software, 14th International SPIN Workshop*, volume 4595 of *Lecture Notes in Computer Science*, pages 263–267. Springer, 2007.

Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.

Amir Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundation of Computer Science*, pages 46–57. IEEE Press, 1977.

Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics*, 14, 2009.

John W. Romein, Aske Plaat, Henri E. Bal, and Jonathan Schaeffer. Transposition table driven work scheduling in distributed search. In *Proceedings of the 16th National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence (AAAI)*, pages 725–731. AAAI Press / The MIT Press, 1999.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall series in artificial intelligence. Prentice Hall, Second edition, 2002.

Viktor Schuppan and Armin Biere. Efficient reduction of finite state model checking to reachability analysis. *International Journal on Software Tools for Technology Transfer*, 5(2–3):185–204, 2004.

Bernhard Steffen. Data flow analysis as model checking. In *Theoretical Aspects of Computer Software (TACS)*, volume 526 of *Lecture Notes in Computer Science*, pages 346–365. Springer, 1991.

Ulrich Stern and David L. Dill. Combining state space caching and hash compaction. In *Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, pages 81–90. Shaker, 1996.

Ulrich Stern and David L. Dill. Parallelizing the murφ verifier. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–267. Springer, 1997.

William J. Stewart. *Introduction to the numerical solution of Markov chains*. Princeton University Press, 1994.

Damian Sulewski, Stefan Edelkamp, and Peter Kissmann. Exploiting the computational power of the graphics card: Optimal state space planning on the GPU. In Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert, editors, *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI Press, 2011.

H. Jaap van den Herik, Jos W. H. M. Uiterwijk, and Jack van Rijswijck. Games solved: now and in the future. *Artificial Intelligence*, 134:277–311, 2002.

Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Logic in Computer Science (LICS)*, pages 332–344, 1986.

Vincent Vidal, Lucas Bordeaux, and Youssef Hamadi. Parallel, Dynamic K-Best-First Search: A Simple but Efficient Algorithm for Multi-Core Domain-Independent Planning. In *International Symposium on Combinatorial Search (SoCS)*, pages 100–107, 2010.

Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

C. Han Yang and David L. Dill. Validation with guided search of the state space. In *Conference on Design Automation (DAC)*, pages 599–604, 1998.

Rong Zhou and Eric A. Hansen. Structured duplicate detection in external-memory graph search. In George Ferguson Deborah L. McGuinness, editor, *Proceedings of the 19th National Conference on Artificial Intelligence (AAAI)*, pages 683–689. AAAI Press / The MIT Press, 2004.

Rong Zhou and Eric A. Hansen. Breadth-first heuristic search. *Artificial Intelligence*, 170(4-5):385–408, 2006.

Rong Zhou and Eric A. Hansen. Parallel structured duplicate detection. In *Proceedings of the 21nd AAAI Conference on Artificial Intelligence (AAAI)*, pages 1217–1222. AAAI Press, 2007.

Rong Zhou and Eric A. Hansen. Dynamic state-space partitioning in external-memory graph search. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI Press, 2011.

Rong Zhou, Tim Schmidt, Eric Hansen, Minh Binh Do, and Serdar Uckun. Edge partitioning in parallel structured duplicate detection. In *International Symposium on Combinatorial Search (SoCS)*, pages 137–138, 2010.