

FAKULTÄT FÜR MATHEMATIK, INFORMATIK UND
NATURWISSENSCHAFTEN DER
RHEINISCH-WESTFÄLISCHEN TECHNISCHEN
HOCHSCHULE AACHEN

Bachelor Thesis in Computer Science

GPU-aided Model Checking of Markov Decision Processes

Philipp Berger

September 2014

submitted to the Lehrstuhl für Informatik 2 in partial fulfillment of the
requirements for the degree of Bachelor of Science at the
RWTH Aachen University

First reviewer:
Prof. Dr. Ir. Joost-Pieter Katoen
Lehrstuhl für Informatik 2
RWTH Aachen

Second reviewer:
Prof. Dr. Erika Ábrahám
Lehrstuhl für Informatik 2
RWTH Aachen

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus fremden Schriften übernommen sind, sind als solche kenntlich gemacht.

Aachen, den 11. September 2014

Declaration

Hereby I declare that I have authored this thesis independently and that I have not used other than the declared sources. I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Aachen, September 11, 2014

Abstract

Probabilistic model checking is a technique for formal verification with successful utilization in the analysis of systems from a diverse background like randomized algorithms, biological processes and communication protocols. This thesis presents an implementation of the well known value iteration algorithm for probabilistic model checking of Markov decision processes in a parallel way on GPGPUs (general purpose graphics processor units). We present a possible preprocessing and grouping using strongly connected components to overcome potential GPU memory limitations and give an implementation for the CUDA (Compute Unified Device Architecture) interface. We show a significant performance gain for several famous benchmark models.

Contents

1. Introduction	6
1.1. Structure	7
1.2. Related Work	8
2. Foundations	9
2.1. Probabilistic Models	9
2.2. Probability Measure on a Markov Chain	12
2.3. Model Checking Reachability Properties on Markov Chains	14
2.4. Markov Decision Processes	16
2.5. Model Checking Reachability Properties on MDPs	19
2.6. Value Iteration	20
2.7. SCCs and Decomposition	22
2.8. General Purpose Computations on GPUs	23
3. Implementation	26
3.1. Analysis & Design Choices - CUDA vs OpenCL	26
3.2. The StoRM Framework	27
3.3. Algorithmic Preliminaries	28
3.4. Sparse Matrix Representations	31
3.5. Our Contribution	32
4. Evaluation	35
4.1. coin case study	35
4.2. leader case study	37
4.3. csma case study	40
4.4. wlan and wlan_collide case study	42
5. Conclusion	44
5.1. Future Work	45
A. Bibliography	47
B. Appendix	50

List of Figures

1.	A transition system for modeling the behavior of a beverage machine. (see Baier et al. [2, page 21])	9
2.	A transition system of a network socket.	10
3.	A Markov chain simulating a die with a coin.	11
4.	Flip-a-coin with either a fair (α) or a biased coin (β).	18
5.	An MDP in which simple action selection fails.	20
6.	A directed graph with four SCCs.	22
7.	Optimal array access pattern by two threads.	24
8.	Blocked array access pattern by two threads	25
9.	An example MDP for Value Iteration.	29
10.	Transition probability matrix with row grouping for figure 9	29
11.	An example of the pyramid-summation-scheme used in the CUDA kernels.	33
12.	Experimental results using the <i>coin</i> models, comparing our implementation against the sequential topological algorithm.	36
13.	Speed-up factor against the sequential topological algorithm and time-share of the parallelized algorithm plotted against the transition count of the <i>coin</i> models.	36
14.	Experimental results using the <i>coin</i> models, comparing our implementation against the StoRM standard algorithm.	37
15.	Speed-up factor against the StoRM standard model checker and time-share of the parallelized algorithm plotted against the transition count of the <i>coin</i> models.	37
16.	Experimental results using the <i>leader</i> models, comparing our implementation against the sequential topological algorithm.	38
17.	Speed-up factor against the sequential topological algorithm and time-share of the parallelized algorithm plotted against the transition count of the <i>leader</i> models.	38
18.	Experimental results using the <i>leader</i> models, comparing our implementation against the StoRM standard algorithm.	39
19.	Speed-up factor against the StoRM standard model checker and time-share of the parallelized algorithm plotted against the transition count of the <i>leader</i> models.	39
20.	Experimental results using the <i>csma</i> models, comparing our implementation against the sequential topological algorithm.	40
21.	Speed-up factor against the sequential topological algorithm and time-share of the parallelized algorithm plotted against the transition count of the <i>csma</i> models.	40

22.	Experimental results using the <i>csma</i> models, comparing our implementation against the StoRM standard algorithm.	41
23.	Speed-up factor against the StoRM standard model checker and time-share of the parallelized algorithm plotted against the transition count of the <i>csma</i> models.	41
24.	Experimental results using the <i>wlan</i> models, comparing our implementation against the sequential topological algorithm.	42
25.	Speed-up factor against the sequential topological algorithm and time-share of the parallelized algorithm plotted against the transition count of the <i>wlan</i> models.	42
26.	Experimental results using the <i>wlan_collide</i> models, comparing our implementation against the sequential topological algorithm.	43
27.	Speed-up factor against the sequential topological algorithm and time-share of the parallelized algorithm plotted against the transition count of the <i>wlan_collide</i> models.	44
28.	Comparing the sparseness of the input matrices against the achieved speed-up with respect to the StoRM standard algorithm.	46
29.	Experimental results using the <i>wlan</i> models, comparing our implementation against the StoRM standard algorithm.	50
30.	Speed-up factor against the StoRM standard model checker and time-share of the parallelized algorithm plotted against the transition count of the <i>wlan</i> models.	51
31.	Experimental results using the <i>wlan_collide</i> models, comparing our implementation against the StoRM standard algorithm.	52
32.	Speed-up factor against the StoRM standard model checker and time-share of the parallelized algorithm plotted against the transition count of the <i>wlan_collide</i> models.	53

1. Introduction

Since the very beginning of computer programming and the emergence of the first bug in 1947, verification of hard- and software has been an ongoing field of study. There has been progress in many directions, some in the way systems are being designed, making design and abstraction a more transparent process. While preventing errors in the first place is a good concept, humans are still prone to fail from time to time. And to make matters worse, techniques like unit tests make errors less likely but can never act as a proof that there are no bugs. Because in certain applications like aerospace or engine control the correctness of circuits is of utmost importance as a single fault may lead to the loss of life or billions in damages, the idea of creating mathematical proofs, such as with the Hoare calculus by Hoare [13] was born. There coexist several promising concepts, as not all are applicable for every scenario. *Model checking* is an approach where via exploring the full state-space of a program properties are proven by viewing all possible executions whereas the Hoare logic typically avoids this by using techniques as finding invariants to describe parts of a program.

Following the model checking approach, a specification gives qualitative properties like “a bad state is never reached” or “all queries are eventually answered”. The model checker takes both the model (or state-graph) and the accompanying specification and verifies each property on the model. If unsuccessful, the model checker not only gives the result of an error but often aides the developers with a counterexample. This helps to understand the problem and whether the problem exists only in the abstracted model or if it reveals a flaw in the original program.

But in many cases, the transition systems used in conventional model checking do not carry enough information to fully capture the nature of certain problems. When moving from state to state, there might be more than one successor, as, depending on input or the modeled process, multiple continuations are possible. To add the knowledge of the probabilities of the successor states, in probabilistic model checking these quantitative aspects are added. Possible scenarios are external input, biological or chemical reactions, all of which typically are modeled using probabilities and non-deterministic decisions.

For modeling probabilistic or non-deterministic choices *Markov decision processes* (MDPs for short) are the most common model type. The resulting model generated from the prototype, the source code or an implementation specification may be only a rough representation but this allows for an incremental approach. Details can be traded in for performance and developers can introduce uncertainty about probabilities directly into the model and verify whether the resulting system would still comply with a given set of rules or requirements.

When checking properties on an MDP, this process often involves computing the probability of reaching a state set, a so-called reachability probability. The

two most notable algorithms for computing these probabilities are firstly *policy iteration* by Howard [16], Van Nunen [34] and Puterman [26], and secondly *value iteration* by Bellman [4] and Shapley [30]. In this thesis only value iteration will be discussed, for further details on policy iteration we refer to Baier et al. [2]. Value iteration is an algorithm that iteratively approximates the target state set reachability in each state by using the current best value from its successor states. The focus of this thesis is on capturing the potential of modern GPUs (graphics processing units) for probabilistic model checking. Since the advent of OpenCL (open computing language) and CUDA (compute unified device architecture) it is obvious that GPUs are suitable for speeding up certain computations, but optimizing traditional algorithms for these environments appears to be a key challenge. In this thesis we present a parallel implementation on GPGPU devices for the value iteration algorithm.

1.1. Structure

In chapter 2 we introduce the theoretical background on probabilistic models and model checking techniques necessary for our implementation. This covers mainly Markov decision processes and accompanying definitions of Markov chains, as well as graph decomposition using strongly connected components. We define probability measurement on MDPs and show that reachability properties, which specify the event of reaching a set of states, are measurable and can be solved using an iterative approach. For further details on the theoretical background we refer to works by Baier et al. [2]. The algorithm and basic ideas used in our implementation are explained alongside the theoretical introductions. For the final part of the foundations we give a short introduction to the specifics of programming general purpose GPUs.

An overview over similar and related work in this area is given in chapter 1.2.

In chapter 3 we explain the details of our implementation and used methods. An analysis and an explanation of our design choices and their respective alternatives is presented in chapter 3.1.

The final part is constituted by the comparison of standard techniques against our parallel implementation on a set of famous benchmark models in chapter 4. We analyze possible factors that influence the speed-up by our methods and derive starting points for future work.

1.2. Related Work

Model checking algorithms and techniques have long been the target of enhancements and improvement. In Holzmann and Bosnacki [15] the authors present parallel adaptations of well known model checking algorithms for the use on multi-core computers. They observe an n -fold reduction in model checking times for systems with n cores and give examples of techniques for which they predict parallelization will not work.

First steps towards GPU aided model checking were done by Bošnački et al. [5] in an extension of the PRISM model checker (see Kwiatkowska et al. [21]). Their work was focused on the acceleration of the multiplication of a sparse matrix with a dense vector using the Jacobi-method to speed up Markov chain model checking.

In Bosnacki et al. [6] the authors built on the earlier work and parallelized more of the algorithms, still using the Jacobi-method.

In 2012 Cormie-Bowins [8] presented a comparison of famous sequential and GPU-aided iterative methods for computing reachability properties on Markov chains. The authors relate to the work by Bosnacki et al. [6].

A study of sparse matrix storage formats and their applicability in GPU-aided sparse matrix-vector multiplication was published by Bylina et al. [7]. The authors compare formats for different levels of “sparseness” in the matrices and give a CUDA-based implementation for Markovian matrices tailored to Markov chains.

GPU-aided non-probabilistic model checking, namely that of LTL (linear temporal logic) properties, was extensively covered by Barnat et al. [3]. They contribute graph-based parallel algorithms that make up a large share of the LTL model checking process.

2. Foundations

In this section we introduce Markov decision processes, an extension of Markov chains by non-deterministic choices between ranges of probabilistic choices. Whereas in transition systems there is a non-deterministic choice between successor states, in Markov chains this is replaced by a probability distribution. To model interleaving or uncertainty, non-determinism is again added in the model of Markov decision processes, in which each state is equipped with a set of probability distributions from which one is chosen non-deterministically.

Intuitively, a state of the underlying transition system represents a snapshot of the modeled process (its state) or a class of equivalent or abstracted states.

All definitions and formality presented here follow those in the definitive book on model checking, *Principles of Model Checking* by Baier et al. [2].

2.1. Probabilistic Models

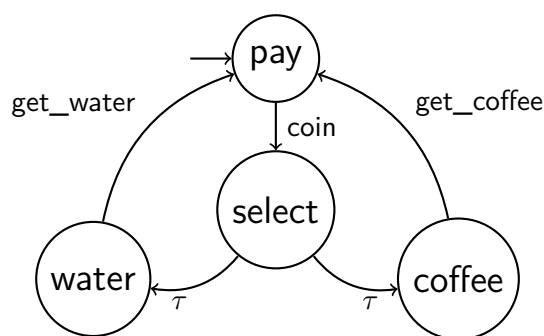


Figure 1: A transition system for modeling the behavior of a beverage machine. (see Baier et al. [2, page 21])

A simple model of a beverage machine is depicted in figure 1 and serves as a very basic description of its functionality. The model consists of four states and some transitions in between. The machine starts in the initial state, moves from state to state following the transitions, on which the actions indicate what action the system performs when moving to the successor state. Each state is labeled by a set of atomic propositions that an observer can see. This means that the observer does not perceive the state the machine is in but the set of atomic propositions which are currently fulfilled. If there is more than one outgoing transition the successor state is chosen non-deterministically.

Definition 1 (Transition Systems). A transition system is defined as a six-tuple $TS = (S, \text{Act}, R, s_{init}, AP, L)$ with

- a set of states S ,
- a set of actions Act ,
- a transition relation $R \subseteq S \times \text{Act} \times S$,
- an initial state $s_{init} \in S$,
- a set of atomic propositions AP ,
- a labeling function $L : S \rightarrow 2^{AP}$.

A transition system is called *finite* iff the set of states S , the set of atomic propositions AP and the set of actions Act are finite.

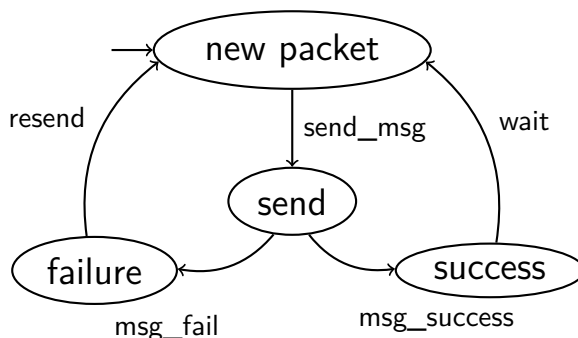


Figure 2: A transition system of a network socket.

However, a transition system lacks the ability to properly model probabilistic characteristics of, for example, a hardware circuit or a network socket. A simple machine which sends a message over a faulty network is depicted in figure 2. Let us assume that from experiments it is known that the probability of a faulty transmission is one in a thousand or 0.1%. But in the model the choice between failure and success is non-deterministic, so sending might always fail (never fail), etc. To combine the knowledge of the probability with the model, we associate a probability distribution with each state such that each successor state has some associated probability of being chosen. This transition system combined with a transition probability function is called a Markov chain. It is very similar to a transition system, but the next successor state is the outcome of a stochastic experiment and not that of a non-deterministic choice.

Definition 2 (Markov chains). A Markov chain is defined as a five-tuple $\mathcal{M} = (S, \mathbf{P}, s_{init}, AP, L)$ with

- a set of states S ,
- a transition probability function $\mathbf{P} : S \times S \rightarrow [0, 1]$ with $\forall s \in S : \sum_{t \in S} \mathbf{P}(s, t) = 1$,
- an initial state $s_{init} \in S$,
- a set of atomic propositions AP ,
- a labeling function $L : S \rightarrow 2^{AP}$.

A state t is called a *successor* of s iff $\mathbf{P}(s, t) > 0$ and a state s is called *absorbing* iff $\sum_{t \in S \setminus \{s\}} \mathbf{P}(s, t) = 0$. The *successor state* of s is chosen probabilistically using the transition probability function \mathbf{P} . Note that the outcome of the stochastic experiment defined by the function \mathbf{P} depends only on the current state and its distribution, not on the path that led there. This is called the *Markov* or *memoryless* property. Many protocols can be modeled as a Markov chain such as the Crowds Protocol by Reiter and Rubin [27] or the bounded re-transmission protocol by D’Argenio et al. [9]. But a common example of what a Markov chain is and how it can be used to model a process is the simulation of a six-sided die by a coin originally proposed by Knuth and Yao [20]. The Markov chain \mathcal{M}_{die} is depicted in figure 3.

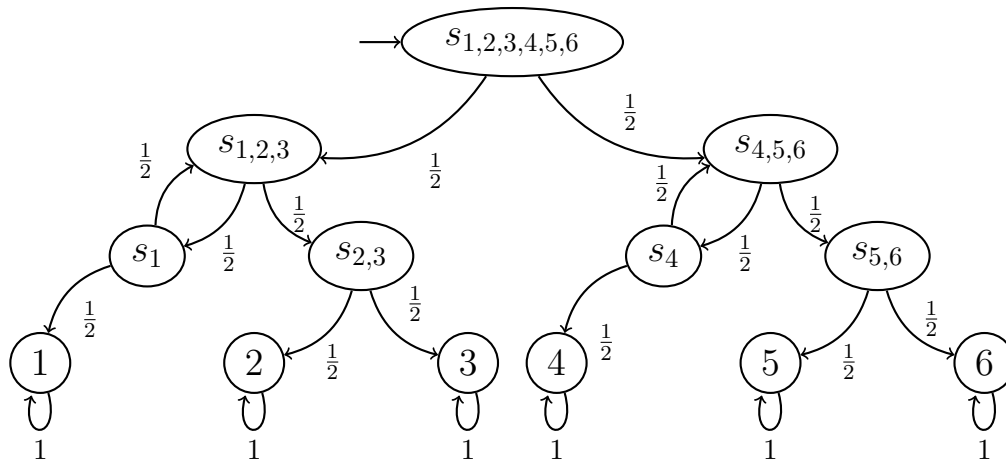


Figure 3: A Markov chain simulating a die with a coin.

The set of states of \mathcal{M}_{die} is

$$S = \{s_{1,2,3,4,5,6}, s_{1,2,3}, s_{4,5,6}, s_1, s_{2,3}, s_4, s_{5,6}, 1, 2, 3, 4, 5, 6\}.$$

The transition probabilities are as given on the transitions in figure 3. A run in the Markov chain starts in the initial state $s_{1,2,3,4,5,6}$. Following the idea of using a fair coin (only heads or tails) to decide progression, from there, two transitions are possible. Both have probability 0.5.

The specification for this process may state that each of the lowest six states representing the result of a roll should have a probability of $\frac{1}{6}$ to be reached from the initial state.

But before questions related to the probability of an event can be properly formulated, a probability measure needs to be defined for a Markov chain. This probability measure associates sets of paths in a given chain with probabilities. The definition of a path follows the intuitive concept of a series of states.

Definition 3 (Paths in Markov chains). *A path in a Markov chain is an infinite state sequence $\pi = s_0s_1s_2s_3\dots \in S^\omega$ such that for all $i \in \mathbb{N}_0$ it holds that $\mathbf{P}(s_i, s_{i+1}) > 0$.*

Let $\text{Paths}(s)$ be the set of all paths starting in state s . A *path fragment* is a finite prefix of a path. Let $\text{Paths}_{fin}(s)$ be the set of all path fragments starting in s .

2.2. Probability Measure on a Markov Chain

Definition 4 (σ -algebras). *A σ -algebra is defined as a pair (Ω, \mathcal{F}) with a non-empty set Ω and $\mathcal{F} \subseteq 2^\Omega$ where Ω is the sample space, \mathcal{F} is the set of events. This pair has to fulfill:*

- $\Omega \in \mathcal{F}$,
- $A \in \mathcal{F} \Rightarrow \Omega \setminus A \in \mathcal{F}$ (therefore also $\emptyset \in \mathcal{F}$),
- \mathcal{F} is closed under countable unions, formally

$$\forall i \in \mathbb{N}_0. A_i \in \mathcal{F} \Rightarrow \left(\bigcup_{j \leq i} A_j \right) \in \mathcal{F}.$$

Definition 5 (Probability Measure). *A probability measure is defined as a function $\text{Pr} : \mathcal{F} \rightarrow [0, 1]$ on a σ -algebra (Ω, \mathcal{F}) with the following constraints:*

- $\text{Pr}(\emptyset) = 0$,
- $\text{Pr}(\Omega) = 1$ (Ω is the almost certain event),

- $\sum_{i \in I_{\mathcal{F}}} \Pr(E_i) = \Pr\left(\bigcup_{i \in I_{\mathcal{F}}} E_i\right)$ with $I_{\mathcal{F}}$ being an index set of \mathcal{F} for pairwise disjoint events (countable additivity property).

Definition 6 (Probability Spaces). A probability space is defined as a three-tuple $(\Omega, \mathcal{F}, \Pr)$ where

- the pair (Ω, \mathcal{F}) is a σ -algebra and
- $\Pr : \mathcal{F} \rightarrow [0, 1]$ is a probability measure on (Ω, \mathcal{F}) .

As denoted above, the elements in \mathcal{F} are called *events*. Paired with the definition of the probability measure, the elements of \mathcal{F} in a probability space are called *measurable events*. Given a Markov chain \mathcal{M} , let $\Omega^{\mathcal{M}} := \text{Paths}(\mathcal{M})$. Intuitively, this means that the sample space of \mathcal{M} is composed of the infinite paths in the chain. The finite path fragments of \mathcal{M} span so called *cylinder sets*. These sets induce a σ -algebra on \mathcal{M} .

Definition 7 (Cylinder Sets). A cylinder set of a finite path fragment is defined to be the set of all infinite paths using the finite path fragment as prefix. Formally: Let $\hat{\pi} \in \text{Paths}_{fin}(\mathcal{M})$.

$$\text{Cyl}(\hat{\pi}) := \{\pi \in \text{Paths}(\mathcal{M}) \mid \hat{\pi} \text{ is a prefix of } \pi\}.$$

On the basis of figure 3 the following example presents the σ -algebra and probability measure of a six-sided die. The sample space Ω is composed of all possible results of a single throw, namely $\Omega = \{1, 2, 3, 4, 5, 6\}$. \mathcal{F} will be the powerset of Ω , which are the 64 possible subsets of Ω .

If Ω is countable, the probability measure is defined as follows:

$$\text{for an event } E \subseteq \Omega \text{ let } \Pr(E) := \sum_{o \in E} (\mu(o)),$$

where μ is the function which assigns each outcome its respective probability. In the case of a die, μ is a constant function with value $\frac{1}{6}$. For example, $\Pr(\emptyset) = 0$, $\Pr(\{1, 3, 5\}) = \frac{1}{2}$.

From Baier et al. [2] we know that the σ -algebra $\mathfrak{A}^{\mathcal{M}}$ associated with a Markov chain \mathcal{M} is the smallest σ -algebra that contains all cylinder sets $\text{Cyl}(\hat{\pi})$ where $\hat{\pi}$ ranges over all path fragments in \mathcal{M} and that there exists a unique probability measure $\Pr^{\mathcal{M}}$ on the σ -algebra $\mathfrak{A}^{\mathcal{M}}$ where the probabilities for the cylinder sets are given by

$$\Pr^{\mathcal{M}}(\text{Cyl}(s_0 s_1 s_2 \dots s_n)) = \mathbf{P}(s_0 s_1 s_2 \dots s_n)$$

where s_0 is an initial state of \mathcal{M} and

$$\mathbf{P}(s_0 s_1 s_2 \dots s_n) := \prod_{i=0}^{n-1} \mathbf{P}(s_i, s_{i+1}).$$

If $n = 0$, e.g. the path fragment consists of one state we define $\mathbf{P}(s_0) = 1$. For paths that do not start in an initial state, formally a new Markov chain \mathcal{M}' is constructed from \mathcal{M} in which the new single initial state is the first state of the path to be evaluated. Intuitively, this approach gives the probability of a path if a certain prefix has already been taken. In some definitions of Markov chains, an initial distribution is used. The approach using only a single initial state is equivalent as both can be transformed to the other by either introducing a new initial state with transitions to all former initial states with the same probabilities as the initial distribution provided, or, for the other direction, by using a distribution which assigns the single initial state the probability one. Consider again the example in figure 3. A possible path fragment in this Markov chain is $\hat{\pi} = s_{1,2,3,4,5,6} s_{1,2,3} s_1 s_{1,2,3} s_{2,3}$. The assigned cylinder set of this path fragment is:

$$\text{Cyl}(\hat{\pi}) = \{s_{1,2,3,4,5,6} s_{1,2,3} s_1 s_{1,2,3} s_{2,3} 2^\omega\} \cup \{s_{1,2,3,4,5,6} s_{1,2,3} s_1 s_{1,2,3} s_{2,3} 3^\omega\}$$

where s^ω denotes the infinite repetition of a state s .

The probability of this cylinder set is given by

$$\mathbf{P}(s_{1,2,3,4,5,6} s_{1,2,3}) \cdot \mathbf{P}(s_{1,2,3} s_1) \cdot \mathbf{P}(s_1 s_{1,2,3}) \cdot \mathbf{P}(s_{1,2,3} s_{2,3}) = \left(\frac{1}{2}\right)^4 = 0.0625.$$

2.3. Model Checking Reachability Properties on Markov Chains

An important part of analyzing a Markov chain in terms of quantitative properties is the question of the probability of reaching a given set of states. This may either be *good* or *bad* states which are or are not to be visited. Let this set of states be called $G \subseteq S$. For this thesis we will use the notation $\diamond G$ as a shorthand for the set $\{\pi \in S^\omega \mid \exists i \in \mathbb{N}_0. s_i \in G\}$. This set contains all those infinite paths fulfilling the property of eventually reaching a G state. To formally calculate the probability of this event we need to write this event as a an intersection or union of cylinder sets. As cylinder sets operate on path fragments, all that is necessary is a set of path fragments from the Markov chain whose cylinder sets represent all paths on which *eventually* G holds. These path fragments all share a basic structure. They start in the initial state of the Markov chain \mathcal{M} , visit some intermediate states which are not in G and then finally reach a state in G . The set $\text{Paths}_{fin}(\mathcal{M}) \cap (S \setminus G)^* G$ contains all such path fragments. As the set of path fragments is countable and our σ -algebra on \mathcal{M} contains all cylinder sets on these fragments, the defined set is measurable. One method of calculating the actual probability of reaching the target states we will explain again using the example in figure 3. Consider the set $G = \{2\}$ containing only the state in which the die rolled the number two. The set of path fragments leading to this state is given by

$$s_{1,2,3,4,5,6} s_{1,2,3} (s_1 s_{1,2,3})^* s_{2,3} 2.$$

The probability of this set is the sum over all paths:

$$\Pr(\diamond G) = \sum_{i=0}^{\infty} \frac{1}{2} \cdot \left(\frac{1}{2} \cdot \frac{1}{2}\right)^i \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8} \cdot \sum_{i=0}^{\infty} \left(\frac{1}{4}\right)^i = \frac{1}{6}$$

We started with a Markov chain modeling a six-sided die. On this model we wanted to compute the probability of the event $\diamond G$, or “eventually the die shows the side with number two”. The result from the last paragraph shows that this probability is indeed one over six as it is expected from a six-sided die. The result therefore confirms that this Markov chain models a six-sided die, at least for the result 2.

Another approach more suitable for automated computation is to solve this by formulating a system of linear equations. Prior to this, the states will be partitioned into one of three sets:

1. S_0 , the set of all states that have no way of reaching a G state.
2. S_1 , the set of all states that almost surely reach G , i.e. with probability one. Note that $G \subseteq S_1$.
3. S_{maybe} , the set of all states that are in neither of the two aforementioned sets.

The partitioning can be algorithmically computed in two steps. In the first step we identify all states that can reach a G state, meaning there exists a path fragment $\hat{\pi} = s_0 s_1 \dots s_n$ where s_0 is the current state and s_n is a state in the set G . For this computation we view the Markov chain as a directed graph. Each state in the Markov chain represents a node in the graph and each transition represents an edge in the graph. We now reverse the direction of the edges in the directed graph, yielding a graph \mathcal{G}' . On \mathcal{G}' we now perform a breadth-first search (Moore [25]) starting from the states in the set G . Since the edges are reversed, the search in a state s enqueues all predecessors of s in the Markov chain. If for a state s in \mathcal{M} there is a path fragment leading to a G state, the backwards breadth-first search from G will at some point process s . Whenever a new state is visited in the search, it is added to the set S_{maybe} if it is not in G .

Formally, we define

$$\text{Pred}(t) := \{s \mid \mathbf{P}(s, t) > 0\}.$$

For a set of states $B \subseteq S$ we define $\text{Pred}(B) := \bigcup_{t \in B} \text{Pred}(t)$. We expand this definition to cover predecessors that are a finite amount of transitions away:

$$\text{Pred}^*(t) := \left\{ s \mid \exists \hat{\pi} = s_0 s_1 \dots s_n. s = s_0 \wedge s_n = t \wedge \bigwedge_{i=0}^{n-1} (s_i \in \text{Pred}(s_{i+1})) \right\}.$$

$\text{Pred}^*(t)$ contains all states from which there exists a path fragment leading to t . This definition can of course be extended to sets of states, too.

The set of all states that have a non-zero probability of reaching the target states and are not target states themselves is then given by

$$S_{\text{maybe}} := \text{Pred}^*(B) \setminus G.$$

The set S_0 is given by $S_0 := S \setminus (S_{\text{maybe}} \cup S_1)$.

In a second step we check if there are states in S_{maybe} which belong into the set S_1 by performing again a backwards breadth-first search from the states in S_0 . If the search visits a G state, it is not explored further. This is to ensure valid results, as it does not matter where a path leads after it has visited a G state. All states in S_{maybe} that have not been visited during the search have no way of not eventually reaching a G state and are moved into S_1 .

Finally for each “maybe” state we introduce a variable $x_s \in [0, 1]$ and an equation

$$x_s = \sum_{t \in \text{succ}(s)} \mathbf{P}(s, t) \cdot x_t$$

where $\text{succ}(s) := \{t \mid t \in S \text{ and } \mathbf{P}(s, t) > 0\}$. For states $t \in S_0$ or $t \in S_1$, the value of x_t in the equation above equals 0 or 1, respectively. The resulting linear equation system can be solved to obtain the reachability probability $\diamond G$ for all states, as described in Baier et al. [2].

2.4. Markov Decision Processes

Protocols where interleaving occurs or when statistical information is unavailable can not be modeled sufficiently as a Markov chain. As in transition systems the ability to perform a non-deterministic choice is necessary to adequately model this features. Non-determinism is able to model three key features:

1. missing statistical information on the probabilities,
2. abstraction of modular subsystems, e.g. interchangeable implementations and
3. interleaving or concurrency of parallel processes.

To extend Markov chains with non-deterministic choices, we introduce *Markov decision processes*, a concept first introduced in the 1950s by Bellman [4] and later on Howard [16]. Transition systems have only non-deterministic choices, Markov chains have only probabilistic choices and Markov decision processes now allow both probabilistic and non-deterministic choices alike.

Definition 8 (Markov Decision Processes). An MDP \mathcal{M} is defined as a six-tuple $\mathcal{M} = (S, \text{Act}, \mathbf{P}, s_{\text{init}}, AP, L)$ with

- a set of states S ,
- a set of actions Act ,
- a transition probability function $\mathbf{P} : S \times \text{Act} \times S \rightarrow [0, 1]$ with $\sum_{t \in S} \mathbf{P}(s, \alpha, t) \in \{0, 1\}$ for all states $s \in S$ and actions $\alpha \in \text{Act}$,
- an initial state $s_{\text{init}} \in S$,
- a set of atomic propositions AP ,
- a labeling function $L : S \rightarrow 2^{AP}$.

An action α is called *enabled* in state s iff $\sum_{t \in S} \mathbf{P}(s, \alpha, t) > 0$.

The function $\text{Act}(s) : S \rightarrow 2^{\text{Act}}$ gives the set of enabled actions in state s . If for a given state s it holds that $|\text{Act}(s)| = 1$, i.e. there is only a single action available, this state has only a probabilistic choice and acts like a state in a Markov chain. Of course, if this holds for all states of an MDP, this specific MDP is equivalent to a Markov chain. It is easy to see that each Markov chain is also a Markov Decision Process. In general, we require that for each state s in an MDP $\text{Act}(s) \neq \emptyset$.

The *successor state* of a state $s \in S$ is chosen in two steps. First, one of the enabled actions in s is chosen non-deterministically, say $\alpha \in \text{Act}(s)$. In the second step, this state now behaves like in a Markov chain since the non-deterministic choice has been fixed, e.g. the successor is chosen using the probability function selected by s and α .

In Markov chains reachability properties (i.e. the probability to reach a certain state or area) could be calculated by solving a system of linear equations. But in MDPs the non-deterministic choices between probability distributions nullify the sense of that question as there is not “a” probability but many.

A developer could ask:

Does this property hold even in a worst-case of non-deterministic choices?

This gives rise to the idea of looking at *maximal or minimal probabilities*. For formalizing how, after having proceeded through the MDP in a particular way, the non-determinism in the current state is resolved, there are many possibilities. The simplest way is to define a static (per-state) assignment which associates a choice with each state. The possibilities range from using an automaton with constant amount of memory to an infinite amount and from using a deterministic function or automaton to using randomness - all these are examples of so-called *policies*.

Definition 9 (Policies). A Policy is defined as a function $\mathfrak{S} : S^+ \rightarrow \text{Act}$ taking a finite path fragment which is mapped onto an enabled action. Therefore for all $\pi = s_0s_1s_2\dots s_n \in S^+$ it holds that $\mathfrak{S}(\pi) \in \text{Act}(s_n)$.

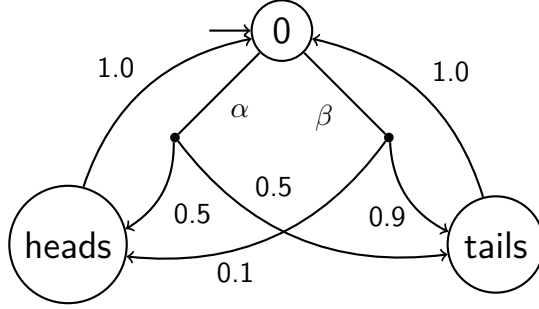


Figure 4: Flip-a-coin with either a fair (α) or a biased coin (β).

The example depicted in figure 4 shows that there does not exist a probability of reaching heads or tails, but several, depending on the resolution of available choices. Other than for example in Markov chains, paths are not only a sequence of states but a sequence of states and the choices taken in the respective state.

Definition 10 (Paths in Markov Decision Processes). A path in an MDP is defined as an infinite sequence of pairs of states and actions $\pi = (s_0\alpha_1)(s_1\alpha_2)\dots \in (S \times \text{Act})^\omega$ such that $\forall i \geq 0. \mathbf{P}(s_i, \alpha_{i+1}, s_{i+1}) > 0$ and $\alpha_{i+1} \in \text{Act}(s_i)$.

We will not define a probability measure on MDPs but rather reuse those of Markov chains. A policy can be applied to an MDP to resolve the non-determinism and receive a Markov chain. This Markov chain is called the *induced Markov chain*. Note that the result is not necessarily finite depending on the used policy. In definition 5 probability measures were defined on Markov chains. To check properties on MDPs we use the probability measure of the induced Markov chains. Let $P := \diamond G$ for $G \subseteq S$ be a reachability property and $\text{Pr}^{\mathcal{M}^\mathfrak{S}}$ the probability measure of the induced Markov chain, then the probability measure of the MDP \mathcal{M} under the policy \mathfrak{S} is defined as:

$$\text{Pr}^\mathfrak{S}(P) = \text{Pr}^{\mathcal{M}^\mathfrak{S}}(P) = \text{Pr}^{\mathcal{M}^\mathfrak{S}}(\{\pi \in \text{Paths}(\mathcal{M}^\mathfrak{S}) \mid \pi \models P\}).$$

Following the previous idea, maximal and minimal probabilities may now be defined by looking at *all* possible policies.

Definition 11 (Minimal and maximal probabilities on MDPs). *Let \mathcal{M} be an MDP and $P := \diamond G$ for $G \subseteq S$ be a reachability property. The minimal resp. maximal probability of P on \mathcal{M} is defined as:*

$$\Pr_{\min}^{\mathcal{M}}(P) = \inf_{\mathfrak{S} \in \text{Policies}} \left(\Pr^{\mathcal{M}_{\mathfrak{S}}}(P) \right)$$

$$\Pr_{\max}^{\mathcal{M}}(P) = \sup_{\mathfrak{S} \in \text{Policies}} \left(\Pr^{\mathcal{M}_{\mathfrak{S}}}(P) \right)$$

where Policies denotes the possibly infinite set of policies on \mathcal{M} .

2.5. Model Checking Reachability Properties on MDPs

In the following, let \mathcal{M} be a finite MDP with state space S , $G \subseteq S$ a set of states and $P_{\mathcal{R}}$ the reachability property $\diamond G$ with the operator $\mathcal{R} \in \{\min, \max\}$. To algorithmically calculate the minimal or maximal reachability probability, the presented definition is not ideal as there are possibly infinitely many policies to consider. In a first step we will justify that for minimal and maximal reachability properties, it is sufficient to only look at a special kind of policies, the *memoryless* policies. These assign to each state an action, independent of the preceding path fragment. For this subclass the definition of the policy can be seen to be a function $\mathfrak{S} : S \rightarrow \text{Act}$, or, conforming to the original definition, the selection of the action depends only on the last state of the path fragment π .

The constructions are very similar, so first we only give the construction for a memoryless policy for a maximal reachability property. The proof of existence is done by starting from a given policy \mathfrak{S}^{\max} which does not have to be memoryless but is one of the policies giving the supremum as in the preceding definition. From there a memoryless policy is assembled. Per assumption we know that for \mathfrak{S}^{\max} the probability measure $\Pr_{\max}^{\mathcal{M}_{\mathfrak{S}^{\max}}}$ returns the maximal probability for the property P . Using this policy we can assign to each state $t \in S$ its maximal probability for reaching a G state under \mathfrak{S}^{\max} :

$$x_t^{\max} = \Pr^{\mathcal{M}_{\mathfrak{S}^{\max}}}(t \models \diamond G).$$

Based on this probabilities we derive for each state a set of possible as maximal actions Act^{\max} . The probability of one state for reaching a G state, if it itself is not a G state, depends on the probability of its successors states and the transition probabilities to them as described in chapter 2.2. Let $(x_s)_{s \in S}$ be the probability vector over all states. By construction each state t has at least one action under which the equation $x_t^{\max} = \Pr^{\mathcal{M}_{\mathfrak{S}^{\max}}}(t \models \diamond G)$ holds. Therefore we define for each state $t \in S$

$$\text{Act}_t^{\max} := \left\{ \alpha \in \text{Act}(t) \mid \left(\sum_{u \in \text{succ}(t)} \mathbf{P}(t, \alpha, u) \cdot x_u = x_t \right) \wedge (t \models \exists \diamond G \text{ under } \alpha) \right\}.$$

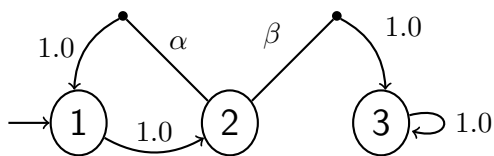


Figure 5: An MDP in which simple action selection fails.

The last condition prevents cases as presented in figure 5, where α would be a viable choice for $x_1 = x_2 = x_3 = 1$, but makes $G = \{3\}$ unreachable. So Act_t^{\max} contains all those actions that are (1) enabled in state t and (2) allow the set G to be reached and (3) provide a maximal reachability probability. A memoryless policy now selects one of these maximal actions. To determine which of the actions in Act_t^{\max} is selected, we define a distance function

$$d(s, \alpha) := \min (|\hat{\pi}| \mid \hat{\pi} = s_0 s_1 \dots s_n, s_0 = s, s_n \in G).$$

This function gives the length of the shortest path fragment leading to G from s under a given action α . If no such path fragment exists, we define the length to be infinity. In a state s we choose the action

$$\alpha := \arg \min_{\beta \in \text{Act}_s^{\max}} (d(s, \beta)).$$

In states that can not reach a G state, an arbitrary action is chosen. \square

The memoryless policy for minimal reachability probabilities can be constructed using a similar approach but we use a policy \mathfrak{S}^{\min} , then define dually to x_t^{\max} the minimal probabilities x_t^{\min} for each state $t \in S$ and define a set Act_t^{\min} . An important difference is that as we now want minimal probabilities, non-reachability of target states is a desirable property. Therefore the last condition regarding the existence of a path to the target states is dropped.

2.6. Value Iteration

As we have established, a memoryless policy is sufficient for calculating minimal or maximal reachability probabilities. For computing the actual reachability probabilities we use value iteration, an algorithm first proposed by Bellman [4]. The idea is to iteratively approximate the local reachability probability by selecting the current best action with respect to the successor state probabilities. This can be characterized as the least fixpoint of a probability transformation function (see Wachter [35]). For maximal reachability probabilities this function

$\mathcal{Y} : [0, 1]^S \rightarrow [0, 1]^S$ is defined as:

$$(\mathcal{Y}(a))_i = b_i := \begin{cases} 0, & \text{if } s_i \in S_0. \\ 1, & \text{if } s_i \in S_1. \\ \max_{\alpha \in \text{Act}(s_i)} (\sum_{t \in S} \mathbf{P}(s_i, \alpha, t) \cdot a_t), & \text{otherwise.} \end{cases}$$

For minimal reachability probabilities, the max operator is replaced by min. This characterization already suggests an iterative approach in which the transformation is applied until the fixpoint is reached. It uses the same preprocessing ideas and state-set partitioning as presented at the end of chapter 2.2, but adapted to the specifics of MDPs. Since the algorithms for calculating minimal and maximal reachability probabilities differ only in a few places, we first present the algorithm for the maximal reachability probabilities. Afterwards, the changes necessary for the version for minimal reachability probabilities are explained. The state-set S is partitioned into the sets S_0 , S_1 and S_{maybe} . For states in the set S_0 under no action there exists a path fragment with a positive probability that leads from the state to the target state set G . Complementary, for states in the set S_1 there exists an action under which there exists no path starting in the state that does not visit a G state eventually afterwards – or the state itself is in G . All other states are in S_{maybe} . These sets can be computed using similar steps as presented for Markov chains in chapter 2.3, except for the last test. Initially a backwards breadth-first search starting from the set S_1 which in the beginning consists only of the G states finds the complement of the set S_0 . To determine whether a state s in S_{maybe} can be moved to the set S_1 , whenever a state is visited during the backwards breadth-first search from S_0 , we keep track of the action under which we entered the state. If for all actions a S_0 state was reachable, it is no S_1 state. This concludes the preprocessing of the state-set. A detailed version of this algorithm and a proof of correctness can be found in Baier et al. [2, pages 853ff].

In each iteration, changes in the local reachability probabilities are propagated one step through the underlying transition system. If for a state the probability value of one of its successor states changed in the last iteration, its own value is updated accordingly, maximizing over the enabled actions which implicitly defines a memoryless policy. Maximizing means that in each step for each state the probabilities under each choice are compared and then the highest is selected as this states current maximal reachability probability. As a direct consequence, a state reachability probability can only increase or remain constant but never decrease.

For minimal reachability properties, the definition of the state partitioning is more involved. The idea is to initially assume all states to be in S_1 and to then iteratively refine the partitioning. For further details we refer to Baier et al. [2].

2.7. SCCs and Decomposition

When reviewing the value iteration algorithm we observe that in most cases it might not be efficient to propagate the probabilities through the whole system directly. If there are circles or similar structures, where the associated probabilities take several steps to converge, propagating the intermediate changes to the outside is a waste. Instead, we need to locate and define components in which the value iteration can be performed independently of the remaining system. One such technique involves finding connected components in the underlying transition system. Cycles are of particular interest as probabilities might propagate around the circle several times before the convergence criterion is fulfilled. Since it would be convenient to only look at each part once, the splitting should take such structural peculiarities into account. In graph theory, a *strongly connected component* (SCC) of a directed graph $G = (V, E)$ is a subset $C \subseteq V$ of the vertices in G such that for each pair of vertices s and t , both t is reachable from s and vice versa. If the set is maximal with respect to G , meaning no vertices can be added without losing the property, the set C is a *maximal* SCC of G .

Definition 12 (Strongly connected components). *A subset C of the vertices of a directed graph $G = (V, E)$ is called an SCC iff $\forall v, w \in C. \exists \pi \in V^+. \exists n \in \mathbb{N}. \text{with } \pi_0 = v, \pi_n = w, (\pi_i, \pi_{i+1}) \in E$. The set is called a maximal SCC of G if additionally it holds that $\forall t \notin C. S' := C \cup t$ is not an SCC.*

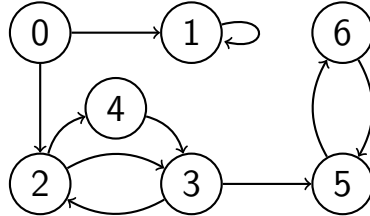


Figure 6: A directed graph with four SCCs.

Consider the directed graph presented in figure 6. There are four maximal SCCs. State 0 forms a so-called trivial SCC as it consists of only one state, same as the SCC containing state 1. The states 2, 3 and 4 form a non-trivial SCC, just as states 5 and 6.

In the following, when talking about an SCC, it is always assumed to be maximal. When referring to MDPs in relation with SCCs, we talk about the directed graph induced by the MDP. Every state is a vertex and there is an edge from vertex v to vertex w iff in the MDP there is a transition from the state associated with

vertex v to the state associated with the vertex w . It is important to observe that given an SCC there exists no path that leaves the SCC and later returns as this would contradict the maximality of the SCC. There are three possible cases for an SCC: There might be both incoming and outgoing transitions, only incoming transitions or only outgoing transitions. In the case of only incoming transitions, the SCC is called a *bottom* SCC. The example in figure 6 has two bottom SCCs, one is state 1 and the other consisting of states 5 and 6. As there is no way of leaving this component, the calculation of the reachability probabilities is independent of the remaining system. If it does not contain a target state the probability is zero for all states, otherwise the value iteration algorithm can be applied to only this subcomponent. For other components an ordering can be defined. If there is an outgoing transition leaving the SCC, the target SCC must be solved first. The maximal SCCs induce a *topological ordering* on the graph. All maximal SCCs of a graph represent a *partition* of said graph. Following the ordering, each SCC can be solved individually as a possibly much smaller instance. Of course it is possible that the entire graph consists of one large SCC or a small number of tiny SCCs together with one large SCC.

The most popular algorithm for computing the maximal SCCs of a graph offering a linear runtime in the number of vertices and edges was published by Dijkstra [10] with recent contributions by Gabow [12]. Other notable algorithms are by Kosaraju and Sharir [31] and Tarjan [32].

2.8. General Purpose Computations on GPUs

While modern mainstream computers with 64-bit CPUs (central processing units) allow for a few parallel threads, GPUs (graphics processing units) use an entirely different architecture. CPUs, often considered to follow the SISD architecture (Single Instruction, Single Data - each instruction is applied once), are heavily optimized for performing well in all situations as they are not tailored to a specific task but need to be able to perform all information processing. GPUs on the other hand were invented as an accelerator for a particular set of tasks, namely graphics processing. In this environment the SIMD architecture (Single Instruction, Multiple Data - a single instruction is applied to several operands) prevailed. A simple use-case of a GPU illustrates this. Consider a view on which for each pixel a transformation has to be applied. A sequence of instructions has to be performed again and again but each time with a different input. To conquer this task, GPUs make use of massive multi-threading where thousands of threads all perform the same actions on their respective data sets. A CUDA-compliant (Compute Unified Device Architecture) GPGPU (general purpose GPU) device is composed of several streaming multiprocessors or SMs. When a kernel is scheduled, a few parameters need to be defined. First of all, grid dimensions need to be set. To better support

processing of 2D and 3D structures, CUDA's topmost organization level consists of a grid. A grid can have one, two or three dimensions. The grid is built from blocks. A block is a collection of threads. Each SM gets assigned a set of thread blocks. These blocks are then split into warps, the basic units of work on the device. A warp consists of 32 threads which always execute simultaneously. The warps are then scheduled for execution. Each thread has a unique identifier. It can be derived from the block id (its position in the grid), the block dimensions (size of a block in each dimension) and the thread id in the current block. For example, the absolute x-coordinate of a thread can be calculated as

$$x := \text{blockIdx}.x \cdot \text{blockDim}.x \cdot \text{threadIdx}.x.$$

Using this formula a running thread localizes itself to decide which work it has to do. Remember that a big workload is to be divided evenly at best between all threads.

The execution of several warps may be interleaved to cover waiting times for memory access. This is due to a design choice that opposed CPU-design, where most commonly huge caches are present to prevent such delays. Global memory access on CUDA devices takes several cycles to complete but meanwhile other warps of the thousands of parallel threads continue execution. For this to work efficiently, operations and data requests have to be extremely symmetric. Branching, conditional execution or even random memory access can lead to large delays and poor performance within a warp. If one thread is divergent, i.e. it does not have the same next instruction as the other threads in the same warp, the whole warp holds its execution and all other threads idle while one thread performs its divergent instructions - which is devastating for the execution efficiency. When accessing shared memory, regardless of reading or writing, for good performance all requests need to coalesce. That means that all threads from a warp request data side by side from each other from a single so-called cache line. A common example is performing a summation on an array.



Figure 7: Optimal array access pattern by two threads.

Possible read patterns are depicted in figure 7 and in figure 8 with the shades signifying which thread is accessing which values. The two threads shown in figure 7 always access two adjacent cells. In the first iteration, thread A uses cell 1 and thread B uses cell 2. Therefore the data required for each cycle could be fetched in a single read. In contrast, the pattern in figure 8 always requires two



Figure 8: Blocked array access pattern by two threads

separate reads (given enough distance to exhaust caches), as in the first iteration thread A uses cell 1 but thread B uses cell 5, which is not adjacent to cell 1. On GPUs these patterns should be optimized to achieve optimal performance, even relating to the order of the data and the ordering of the threads. If for example a read operation can not be performed as a coalesced memory access, then each thread will execute for itself with the others waiting for their turn. In a single warp this means that the overall performance is at most $\frac{1}{32}$ of the maximal performance possible.

Another common problem is called *global synchronization*. This occurs whenever all threads of a kernel need to synchronize and may only continue once all threads have reached a certain point. When implementing iterative algorithms the convergence criterion usually requires this kind of synchronization, as whether or not a next iteration should be calculated depends on the complete result. Additionally, in algorithms like value iteration where the input of the next iteration consists of the result of the last, the next computation should not start before the preceding iteration has finished executing. Another way to implicitly achieve this is to start one kernel per iteration. This comes at the cost of the kernel-invocation time overhead but drastically simplifies the program structure. The specific hardware and configuration of the GPGPU devices like cache sizes, number of streaming multiprocessors, number of threads per warp, etc. changes with each generation and is being abstracted through the CUDA (*Compute Unified Device Architecture*) interface. It comes with an extension to the C++ programming language giving access to the full capabilities of the devices. NVIDIA provides additional libraries like Thrust, a library of parallel algorithms and data structures (Hoberock and Bell [14]) that allows easy utilization of common algorithms.

3. Implementation

Algorithms for *Value Iteration* are known and implemented in countless tools (e.g. Prism (Kwiatkowska et al. [21]), MRMC (Katoen et al. [19])). Most of the implementations are designed for execution on CPUs, but there exist a few other that focus on GPUs. There has been some effort on re-implementing the whole model checking process in a GPU-friendly way, but since the aforementioned problems of *global synchronization* this may not be the optimal approach with current hardware generations.

The focus in this thesis was on finding those parts of the algorithm that may be executed efficiently on a GPU and deciding on necessary preprocessing steps. Most implementations can only work if the complete working set (probability matrix, result vectors) fits into the main memory. For CPU-bound implementations this is no real drawback since computing power is a much more limiting factor here than main memory sizes which currently tend to be in the area of hundreds of gigabytes on industrial workstations. On GPGPUs however, ten gigabytes or less is common. Therefore we evaluated and used a preprocessing step known as *strongly connected component decomposition*. As shown in chapter 2.7, one can compute the induced *topological ordering* of the SCC decomposition. The SCC decomposition itself was not parallelized in this thesis.

The approach via SCCs enables GPU aided computation on models larger than the available accelerator card memory - of course only in the common case where there is not one SCC nearly covering the whole MDP.

3.1. Analysis & Design Choices - CUDA vs OpenCL

Ever since the field of *GPGPU* or general-purpose computing on graphics processing units took off, the two main competitors on the market of GPGPU accelerator cards, AMD and NVIDIA, have been fighting over market shares, not only by means of new hardware but also by means of offering tools and specific languages that allow optimal performance on their devices to tie a customer to their brand. As the decision for a framework is also a decision for target hardware, we present a list of pros and cons.

OpenCL

As an open standard OpenCL has first appeared in 2009. It is being maintained by a non-profit organization and has been adopted by several large companies such as AMD, Intel, Apple and NVIDIA. OpenCL offers a very open environment supporting heterogeneous platforms including but not limited to CPUs, GPUs, DSPs (digital signal processors), and even FPGAs (field-programmable gate arrays). Because of the number of supporters, many vendors and developers support

OpenCL. A downside comes as the results of its heterogeneous platforms - since its not very device specific, low-level programming or the use of highly specialized routines are not possible. The fact that OpenCL applications can even run on Android mobile devices is irrelevant to the targets of this work.

CUDA

CUDA was first released in 2007. It is very similar to OpenCL, though the hardware abstraction layers differ in terminology and coarseness. It was designed by NVIDIA to advance the usage of GPGPU computations and allow better and easier access to device functions by developers. The specific targeting of GPUs allows a better focus on available resources. The compiler is updated with each new hardware generation to integrate new features. NVIDIA's main target audience of CUDA is the scientific computing community. Therefore many ready-made and optimized libraries are available both from NVIDIA and from other companies, targeting the most common tasks in scientific computations.

There have been some studies and performance evaluations in the past comparing OpenCL and CUDA mainly for performance, in which the general consensus is that CUDA outperforms OpenCL in most use cases, but the difference is thin (Karimi et al. [18], Fang et al. [11]).

Our decision towards CUDA is based on several points. Firstly, since this implementation's target is a higher speed and not its portability to other platforms, the better performance of CUDA is a plus. Secondly, we found the interfaces to be very similar and assume that porting our current implementation to the OpenCL interface is relatively straightforward. Thirdly, the availability of free and open-source tools for scientific computing using the CUDA interface allowed for an easy and smooth transition from the conventional style of programming to the highly optimized and sophisticated GPU algorithms. Fourthly, it seems that NVIDIA won the battle for scientific computing. Judging from the amount of CUDA-capable devices on recent Top500 Supercomputer lists and the availability of such resources at the clusters of RWTH Aachen, it is very unlikely that NVIDIA and with them CUDA will vanish anytime soon.

3.2. The StoRM Framework

The framework for our implementation was the StoRM project (Stochastic reward model checker). It is the successor of the MRMC project by Katoen et al. [19], but still unpublished and under development. It can be downloaded from our Git repository¹. StoRM offers support for parsing various types of models like DTMCs, MDPs, CTMCs and CTMDPs (see Baier et al. [2]) with ready-made interfaces for implementing new techniques. On the basis of an existing MDP

¹<https://sselab.de/lab9/private/git/storm>

model checker we implemented a parallel version of the value iteration algorithm for solving reachability probability problems.

3.3. Algorithmic Preliminaries

Starting a kernel (a program) on the GPU comes with an overhead as it takes some time to synchronize resources, prepare the program for execution, etc. As computing the SCC decomposition of a model is an unnecessary overhead while it is small enough to fit into the GPU memory, we therefore decided to use the SCC decomposition only if the model is too large to fit into the GPU memory. This hardware-dependent size constraints are identified at runtime, since they depend on current utilization. Additionally, we employ a small optimization heuristic. Whenever a model is determined to be too large and is decomposed into SCCs, we use a maximum-fit strategy to re-join SCCs. Thereby we follow the topological ordering of the SCCs, beginning from the currently topmost SCC, i.e. the SCC that has to be solved next. While the available GPU memory permits it, we join the SCC next in line with the topmost. This upholds all requirements since all prerequisites have either been solved already or will be solved in the same step. Should the case arise that a single SCC is too large for the GPU memory, the algorithm falls back to a CPU version to solve this component individually.

Algorithm 1. *Calculating the maximal reachability probability $\mathbf{P}_{max}^{\mathcal{M}}(s \models \diamond G)$*

Input: Markov decision process $\mathcal{M} = (S, \mathbf{P}, s_{init}, AP, L)$, target states $G \in S$

Output: A probability vector $(x_s)_{s \in S}$

```

1 begin
2    $(x)_{s \in S} \leftarrow 1.0$  for  $s \in G$ , 0.0 else;
3   repeat
4     for  $s \in S_{maybe}$  do
5        $x'_s \leftarrow \max \{ \sum_{t \in S} \mathbf{P}(s, \alpha, t) \cdot x_t \mid \alpha \in \text{Act}(s) \}$ ;
6     end
7   until  $\max \{ |x_s - x'_s| ; s \in S \} \leq \text{required precision}$ ;
8   return  $(x_s)_{s \in S}$ ;
9 end

```

A pseudo-code implementation of the value iteration algorithm is presented in algorithm 1. Implementation-wise, we focused on the fixpoint characterization which leads to the algorithm explained in the following. The transition probability function of an MDP can be represented using a transition probability matrix. Its basic form is $S \times S$, but since the non-deterministic choices need to be considered, the matrix is extended to $(S \times \text{Act}) \times S$. For each state s and choice α , there is a

row in this matrix representing all outgoing transitions with each cell in a column t containing the probability value $\mathbf{P}(s, \alpha, t)$.

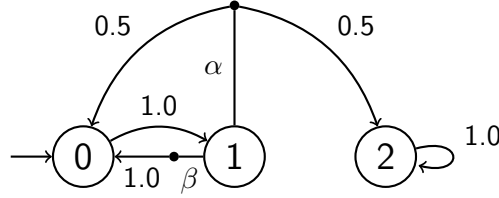


Figure 9: An example MDP for Value Iteration.

The transition probability matrix for the example presented in figure 9 is shown in figure 10.

$$\text{choices } \alpha \text{ and } \beta \text{ in state 1 } \left\{ \begin{array}{ccc} (0.0 & 1.0 & 0.0) \\ (0.5 & 0.0 & 0.5) \\ (1.0 & 0.0 & 0.0) \\ (0.0 & 0.0 & 1.0) \end{array} \right.$$

Figure 10: Transition probability matrix with row grouping for figure 9

A row in the given matrix represents a probability distribution of one state under a choice. The lines in the matrix denote how many rows (choices) there are for a state. For example, in state 1 there are two choices. In the matrix, rows 1 and 2 are grouped together, forming the choices of state 1 – assuming a zero-based indexing. The columns in the matrix give the probability to go to state 0, 1 or 2, respectively. In this example, the target state set G contains only state 2. We want to compute the probability $\mathbf{P}_{max}^{\mathcal{M}}(s \models \diamond G)$. The vector $(x_s)_{s \in \mathcal{S}}$ is multiplied with the transition probability matrix. Note that the vector initially contains zeros in all entries except those corresponding to G states, where its value is one. Following this example, the initial vector x is $(0.0, 0.0, 1.0)^T$. This multiplication results in a new vector that contains the current reachability probability for each state under one chosen action. The first step results in:

$$\begin{pmatrix} 0.0 & 1.0 & 0.0 \\ 0.5 & 0.0 & 0.5 \\ 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{pmatrix} \cdot \begin{pmatrix} 0.0 \\ 0.0 \\ 1.0 \end{pmatrix} = \begin{pmatrix} 0.0 \\ 0.5 \\ 0.0 \\ 1.0 \end{pmatrix}$$

This vector now contains probabilities for all available pairs of states and actions. It follows the same row grouping as the used matrix. We now maximize over the actions as we are interested in the maximal reachability probability. In the example, there is only one state with more than one enabled action, namely state 1.

$$x'_1 = \max \{0.5, 0.0\} = 0.5.$$

For all other states the maximum over one value returns the value itself. This way the vector x' for the next iteration is build. Now the entry-wise difference between the vectors can be calculated as shown in the pseudo-code. If the greatest absolute difference is below the chosen precision, the algorithm terminates as the calculation has *converged*. Otherwise the next round begins with x' as the new x vector. This approach has the prerequisite that all G states have only a self-loop with probability one. In other cases, the algorithm may loose the information that this G states already have the fixed probability one. Therefore, in a preliminary step we take the submatrix C of A consisting of only the states in S_{maybe} . Remember that for a partitioning of the state set S , the partition S_{maybe} contains all those states that have a positive probability of reaching a G state - either under some action (maximal reachability) or under all actions (minimal reachability). To compensate for the loss of the S_1 (and S_0) states, we add a new vector b . This vector describes for each state the probability of reaching a S_1 state in a single step, e.g. if a state has a G state as a direct successor, the transition probabilities to all those successors are added up.

$$(v)_{s \in S_{maybe}, \alpha \in \text{Act}(s)} := \sum_{t \in (\text{succ}(s) \cap G)} \mathbf{P}(s, \alpha, t).$$

Similar to taking a submatrix of A , the vector x is reduced to contain only the states in S_{maybe} , too. We then define the iterative equation for this system as

$$x' = C \cdot x + b$$

which for $s \in S_{maybe}$ and $\alpha \in \text{Act}(s)$ can also be written as

$$(x')_{s, \alpha} := \sum_{t \in (\text{succ}(s) \cap S_{maybe})} \mathbf{P}(s, \alpha, t) \cdot x_{t, \alpha} + \sum_{t \in (\text{succ}(s) \cap G)} \mathbf{P}(s, \alpha, t).$$

In other words we split those states with static probability values from those with dynamic values.

A pseudo-code version of the used iteration scheme is depicted in algorithm 2.

Algorithm 2. Calculating the reachability probability $\mathbf{P}^M(s \models \diamond G)$

Input: Transition probability submatrix of an MDP C over all states
 $s \in S_{\text{maybe}}$, static probability vector $b \in [0, 1]^n$, initial state probability
vector $x \in [0, 1]^n$, row-grouping information R_C on C , reduce operator
 $\mathcal{R} \in \{\min, \max\}$, required precision ε

Output: A probability vector $(x_s)_{s \in S_{\text{maybe}}}$

```

1 begin
2   repeat
3     mulRes  $\leftarrow C \cdot x + b$ ;
4     for group  $\in R_C$  do
5        $x'_{\text{group}} \leftarrow \mathcal{R}_{s \in \text{group}}\{\text{mulRes}_s\}$ ;
6     end
7     swap( $x, x'$ );
8   until  $\max\{|x_s - x'_s| \mid s \in S\} \leq \text{required precision } \varepsilon$ ;
9   return  $x$ ;
10 end

```

3.4. Sparse Matrix Representations

The multiplication step has been further optimized. For most applications a dense matrix representation is used. Dense refers to how entries in the matrix are stored in the memory, in this case for a matrix of size $m \times n$ there will be $m \cdot n$ memory cells containing the values. Often, because of how a specific problem is build, there are more efficient ways to store a matrix. Consider cases like symmetric or diagonal matrices – they allow for much more space efficient representations. In model checking the specialty is the low quantity of entries that differ from zero. Remember that in model checking we use probability matrices. A transition to another state exists iff the corresponding matrix entry is not zero and thus greater than zero. A typical trait of a model is that from a state only a few other states can be reached. It seems counter intuitive that there is a state from which all other states are directly reachable. Therefore each row representing the outgoing transitions of a state contains a small number of entries that differ from zero. For example in the case study of the `coin` protocol, the instance `coin8_2` has 61,018,112 states but only 403,856,384 transitions. Only 0.000010847% of all entries in its transition probability matrix contain a non-zero value. Such matrices are called *sparse* as they are sparsely populated.

We use a storage format which was made distinctly for this kind of matrix called CSR (compressed sparse row). For other formats or algorithms optimized for sparse matrices, see Tewarson [33]. A given matrix is encoded into three arrays. The first array a_{val} contains all values row by row. A second array a_{col} contains for

each entry in a_{val} the index of the column where this value is located in. Lastly, the array a_{row} describes which entries in a_{val} are in which row. Instead of saving the row index for each entry, the start and end index for each row in a_{val} are saved.

Consider again the matrix

$$\begin{pmatrix} 0.0 & 1.0 & 0.0 \\ 0.5 & 0.0 & 0.5 \\ 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{pmatrix}.$$

In CSR format, this matrix can be encoded as

$$a_{val} = \begin{pmatrix} 1.0 \\ 0.5 \\ 0.5 \\ 1.0 \\ 1.0 \end{pmatrix}, a_{col} = \begin{pmatrix} 1 \\ 0 \\ 2 \\ 0 \\ 2 \end{pmatrix}, a_{row} = \begin{pmatrix} 0 \\ 1 \\ 3 \\ 4 \\ 5 \end{pmatrix}.$$

When looking for the value of an entry $(A)_{1,1}$, we start in a_{row} . The entry with index 1 (using zero-based indexing) in a_{row} is 1. The next entry is 3. We now know that the requested row spans two entries in a_{val} and a_{col} – the entries 1 and 2. As requested we traverse the a_{col} array for the column index 1. Since at position 1 the column index 0 is lower than 1, we proceed to the next entry. At position 2 the index 2 is larger than 1, we therefore conclude that the requested value is not in our representation and is, per definitionem, 0.

3.5. Our Contribution

For the sparse matrix-vector multiplication we use an algorithm based on the CUSP library. It assigns one warp per row, therefore 32 threads share a single row. Each thread starts by computing its global thread id. After that, the thread lane is computed. The thread lane describes the local id of the thread in its warp. As there are several warps in a block, this is not equal to the value of `threadIdx.x`. Then the row index i_{row} this thread works on is computed by dividing the global thread id by the number of threads per warp.

In the next step the pointers from the a_{row} array are fetched, both the entry i_{row} and the entry $i_{row} + 1$ since the latter marks the end of the row. For performance reasons, the first thread in the warp fetches in value at i_{row} and the second thread the value at $i_{row} + 1$. These two values are then written to a warp-local cache, therefore all threads of the warp now have access to them. Now all threads compute their starting index in the a_{val} and a_{col} arrays by adding the row start index with the thread lane index. Remember that the thread lane describes the position of

the thread in the warp. In an example with four threads forming a warp and a row start index of 42, the first thread starts on index 42 and the fourth starts on index 45.

With the setup completed, the threads in the warp proceed through the row in strides of the warp size. In each iteration each thread reads a column index and the corresponding matrix entry, load the accompanying value from the vector x (at the column index position) and multiply the matrix entry with the vector entry. The result is added to a thread-local summation variable. After the row has been crossed, the local sums need to be added up to form a warp- and therefore row-sum. This is done via a pyramid-scheme as depicted in figure 11.

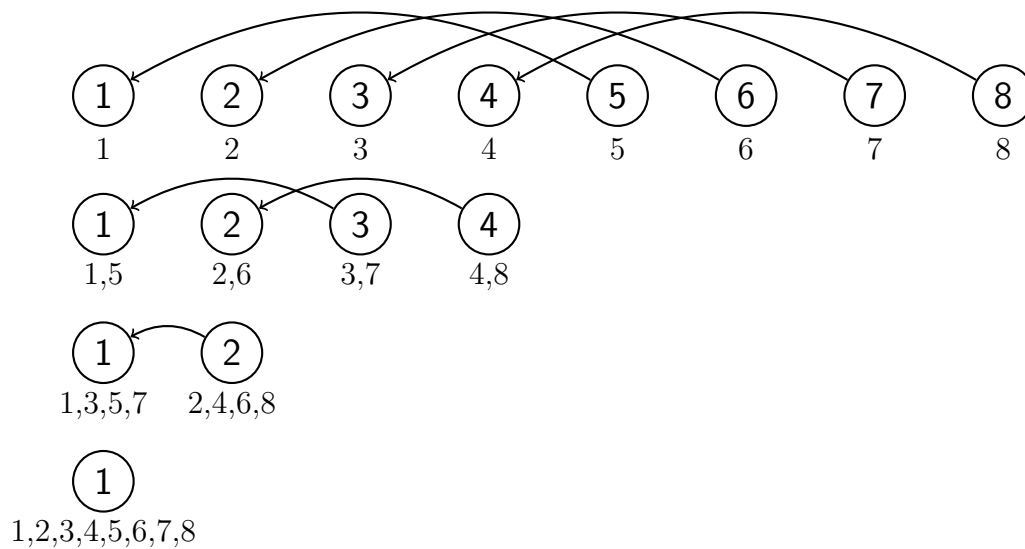


Figure 11: An example of the pyramid-summation-scheme used in the CUDA kernels.

While the figure shows a simplified version where only a part of the threads perform a copy and summation in each step, the kernel performs the same steps on each thread, as otherwise the divergence would impose an unnecessary slowdown. This implies that the result array has 150% of the required size, enabling in the example even the thread with number 8 to copy and add the value from the non-existent thread 12. This value is initialized to zero and therefore does not change the result but all threads can perform the same operations without leaving the legal array bounds. In the last step, each thread has the sum over all threads on its right side plus itself. Concluding the computation the first thread writes its aggregated sum back to global memory into the result vector.

Following the algorithm presented in listing 2, we performed the first half of line 3. The required addition of the vector b is performed using the Thrust primitives library (vector operation with operator *plus*).

Lines 4 – 6 are more involved since the row-grouping of the matrix imposes how the entries in the *mulRes* vector will be reduced. Remember that we minimize or maximize (depending on whether we compute minimal or maximal reachability properties) over all enabled actions in a state. Each such enabled action in a state comes with a row in the transition probability matrix and therefore produces an entry in the *mulRes* vector. We implemented a kernel for this based on the sparse matrix-vector multiplication kernel presented earlier. Instead of traversing a row of a matrix each warp traverses all choices in a state (we assign a warp per state). The setup is performed as in the multiplication kernel, but this time based on a vector describing the row grouping. The row grouping vector contains an entry for each state (plus an additional entry describing the end of the very last group). Each entry is the index of the first row of the corresponding state. The row grouping matrix of the example matrix presented earlier is $(0, 1, 3, 4)^T$. Note that the last entry acts as a pivot element - there is no fourth state and no row with index 4, but an algorithm can easily determine the amount of enabled actions in each state by subtracting the starting index of the next higher state from the starting index of the current state. Each thread of a warp now calculates a local minimum or maximum. In each step, it compares the current minimum or maximum against the value in the *mulRes* vector at its current index. After all actions of a state have been visited the pyramid-scheme is used again to reduce the thread local extremum to a warp-wide extremum. Concluding the computation the first thread writes the computed extremum back to global memory into the result vector.

The final part of the algorithm presented in listing 2 consists of checking convergence. This involves the pyramid-scheme yet again. In a first step, the difference between the newly computed x' vector and the x vector from the last iteration is computed. We support both relative and absolute comparison. We defined a custom, optimized CUDA binary transformation operator taking two arguments and returning a single positive value. This transformation operator is applied to each entry of the difference vector. After subtracting the global maximum of the computed difference needs to be found. Such an operation is called as mentioned earlier a *reduce* operation. For the simple case of a vector-wide reduce, the Thrust library again provides a solution. The iteration finishes with the comparison of the computed maximum against the specified precision, i.e. the maximum over the absolute differences between the old and new solution vector x is compared to a given precision value, for example $1 \cdot 10^{-6}$. Either a next iteration is begun or the algorithm terminates.

4. Evaluation

All experiments were performed on a machine running Windows 7.1 64-bit with an Intel Core i7 processor, 32 GB RAM and a NVIDIA GeForce GTX 670 with 2 GB video memory. The software was compiled from C++ code using Microsoft Visual Studio 2013 SP2. For the experiments several famous benchmark models were used. We compared our GPU aided value iteration algorithm against two CPU-based competitors. The first competitor is the sequential version of the algorithm we implemented. For the second test we wanted to see how we compare against the standard StoRM model checker algorithm which is implemented using Intel Threading Building Blocks, a library for CPU based parallel algorithms. For each benchmark we present a table giving an overview over (a) the used instances in column “example”, (b) state- and transition count of the instance the respective columns, (c) average measured runtime of the standard model checker using the CPU solver and our CUDA/GPGPU based model checker and (d) the factor between the two foregoing times. A factor larger than one means that our implementation performed better than the standard model checker, a factor less than one implicates a slowdown. To better illustrate the relationship between the size of the model and the possible speedup, we present, in addition to a table summarizing the experimental results, a plot of the speed-up factor and the time spent in the GPU solver in percent against the number of transitions.

4.1. coin case study

Firstly, we used the randomized consensus shared coin protocol by Aspnes and Herlihy [1] and also discussed in Segala [29]. The protocol models n asynchronous processes that, starting from an initial value of 1 or 2 provided by the environment, proceed through rounds. In each round the processes read the current decision of the others and check whether they agree. The agreement attempt is called a random walk, which can be understood as a Markov decision process using both deterministic and non-deterministic choices. Should no agreement be reached, using a shared coin protocol each process decides on its next value. The experimental results for the coin protocol are presented in figures 12 to 15.

The example *coin8_2* is too large to fit into the memory of the used GPGPU device, therefore it is first split into SCCs and then solved iteratively. This leads to a slightly worse speed-up factor for this instance. We observe that for small instances with around a thousand transitions the overall runtime is worse when compared to the CPU based model checker. For larger instances greater than 1500 transitions, even though the time share of our solver decreases with increasing model size, the speed-up continually increases apart from minor fluctuations and except for the largest instance when comparing to the StoRM standard model

example	states	transitions	$\frac{\text{transitions}}{\text{states}}$	time (CPU)	time (CUDA)	factor
coin2_2	272	492	1.809	35 ms	300 ms	0.117
coin2_4	528	972	1.841	100 ms	655 ms	0.153
coin2_6	784	1452	1.852	263 ms	1265 ms	0.208
coin4_4	43136	144352	3.346	5313 ms	2955 ms	1.798
coin4_6	63616	213472	3.356	13022 ms	7010 ms	1.858
coin6_4	2376448	11835456	4.980	323066 ms	198890 ms	1.624
coin6_6	3494656	17434176	4.989	928068 ms	550555 ms	1.686
coin8_2	61018112	403856384	6.619	30532249 ms	4052731 ms	7.534

Figure 12: Experimental results using the *coin* models, comparing our implementation against the sequential topological algorithm.

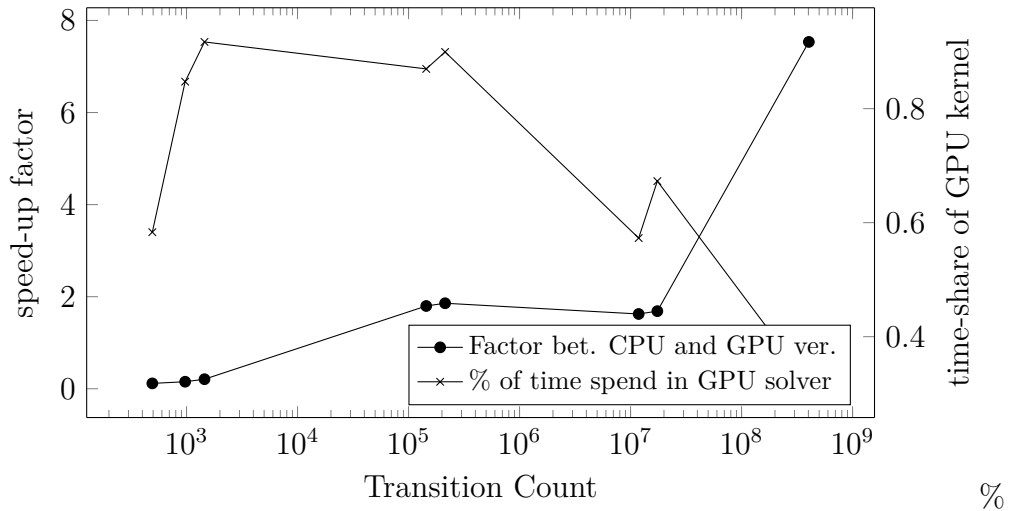


Figure 13: Speed-up factor against the sequential topological algorithm and time-share of the parallelized algorithm plotted against the transition count of the *coin* models.

checker. The speed-up of only the solving step is therefore much larger than visible from the table. This is due to the parallel algorithm taking only a low fraction of the the total execution time.

example	states	transitions	$\frac{\text{transitions}}{\text{states}}$	time (CPU)	time (CUDA)	factor
coin2_2	272	492	1.809	30 ms	300 ms	0.100
coin2_4	528	972	1.841	55 ms	655 ms	0.084
coin2_6	784	1452	1.852	121 ms	1265 ms	0.096
coin4_4	43136	144352	3.346	4010 ms	2955 ms	1.357
coin4_6	63616	213472	3.356	12835 ms	7010 ms	1.831
coin6_4	2376448	11835456	4.980	684665 ms	198890 ms	3.442
coin6_6	3494656	17434176	4.989	2124812 ms	550555 ms	3.859
coin8_2	61018112	403856384	6.619	9568588 ms	4052731 ms	2.361

Figure 14: Experimental results using the *coin* models, comparing our implementation against the StoRM standard algorithm.

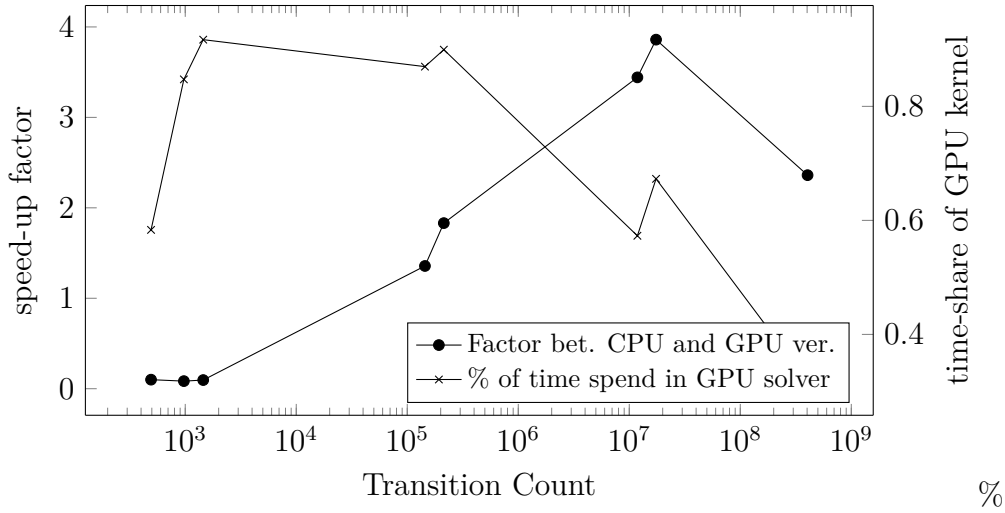


Figure 15: Speed-up factor against the StoRM standard model checker and time-share of the parallelized algorithm plotted against the transition count of the *coin* models.

4.2. leader case study

Secondly, we chose the asynchronous leader election protocol by Itai and Rodeh [17]. It models n processors that communicate over an asynchronous ring trying to elect a leader (a distinct processor). In each round the active processors either stay active or become inactive based on a probabilistic choice and the choice of its

neighbor on the ring. The protocol terminates when only a single processor is left as being active. When reviewing execution times in figure 16, we suspect that the extremely high execution times and speed-ups in the instances `leader6`, `leader7` and `leader8` under the sequential topological algorithm result from a bug in the StoRM framework or measurement error.

example	states	transitions	$\frac{\text{transitions}}{\text{states}}$	time (CPU)	time (CUDA)	factor
<code>leader3</code>	364	654	1.797	25 ms	365 ms	0.068
<code>leader4</code>	3172	7144	2.252	105 ms	310 ms	0.339
<code>leader5</code>	27299	74365	2.724	1221 ms	395 ms	3.091
<code>leader6</code>	237656	760878	3.202	100920 ms	1645 ms	61.350
<code>leader7</code>	2095783	7714385	3.681	8823543 ms	15685 ms	562.547
<code>leader8</code>	18674484	77708080	4.161	T.O. (> 48h)	251271 ms	687.704

Figure 16: Experimental results using the *leader* models, comparing our implementation against the sequential topological algorithm.

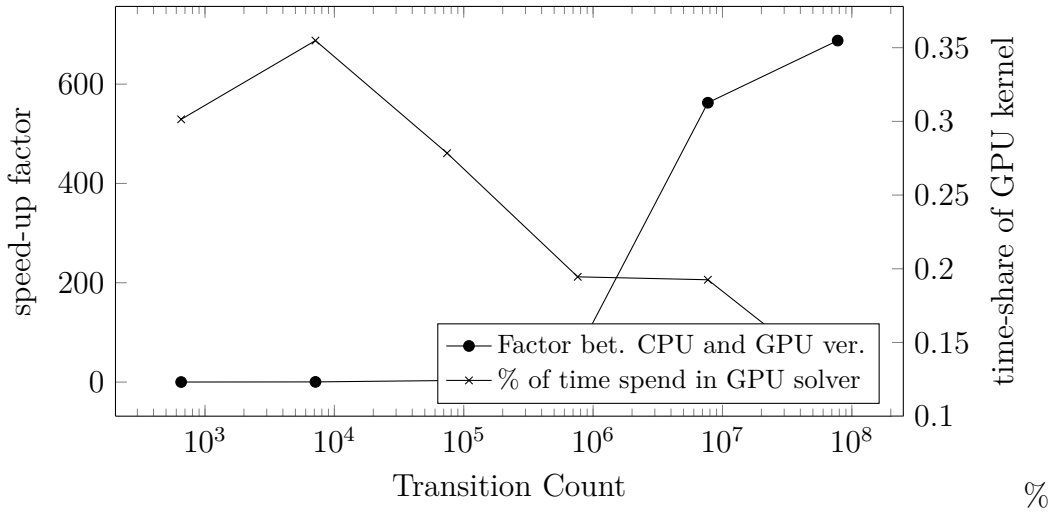


Figure 17: Speed-up factor against the sequential topological algorithm and time-share of the parallelized algorithm plotted against the transition count of the *leader* models.

We refer to figure 18 for more reliable results on the *leader* case study. Similar as `coin8_2`, the example instance `leader8` is too large to fit into the memory

example	states	transitions	$\frac{\text{transitions}}{\text{states}}$	time (CPU)	time (CUDA)	factor
leader3	364	654	1.797	15 ms	365 ms	0.041
leader4	3172	7144	2.252	35 ms	310 ms	0.113
leader5	27299	74365	2.724	290 ms	395 ms	0.734
leader6	237656	760878	3.202	3595 ms	1645 ms	2.185
leader7	2095783	7714385	3.681	41580 ms	15685 ms	2.651
leader8	18674484	77708080	4.161	471045 ms	251271 ms	1.875

Figure 18: Experimental results using the *leader* models, comparing our implementation against the StoRM standard algorithm.

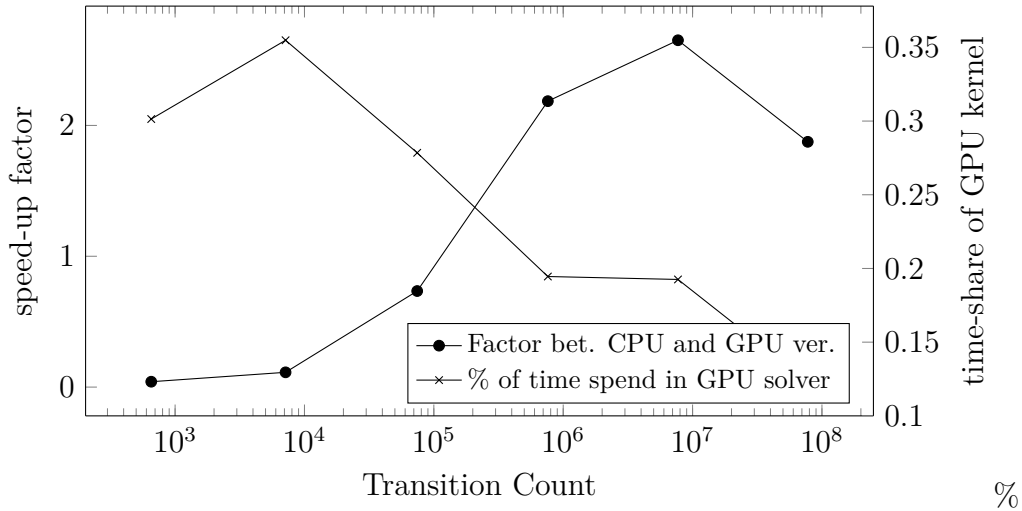


Figure 19: Speed-up factor against the StoRM standard model checker and time-share of the parallelized algorithm plotted against the transition count of the *leader* models.

of the used GPGPU device, therefore it is first split into SCCs and then solved iteratively. This preprocessing step results in a small break in efficiency. Aside from small models with less than 10^4 states we observe a continuous decrease in the percentage of time spent in our solver. The speed-up increases nevertheless with increasing model size. Therefore our implementation performs exceedingly well on this case study.

4.3. csma case study

Thirdly, we present a case study by Kwiatkowska et al. [23] (and Kwiatkowska et al. [24]) concerning the IEEE 802.3 CSMA/CD (carrier sense, multiple access with collision detection) protocol. It is used on single-channel wired networks where several stations need to communicate without crosstalk. In the model n stations each try to send a message. When a message is enqueued, the station first listens on the medium and then, when free, starts sending. In the case there is another transmission, a random waiting time passes before the stations retries. Because of the large memory requirements only the instances up to `csma3_4` could be tested.

example	states	transitions	$\frac{\text{transitions}}{\text{states}}$	time (CPU)	time (CUDA)	factor
<code>csma2_2</code>	1038	1282	1.235	30 ms	155 ms	0.194
<code>csma2_4</code>	7958	10594	1.331	215 ms	330 ms	0.652
<code>csma2_6</code>	66718	93072	1.395	1785 ms	2015 ms	0.886
<code>csma3_2</code>	36850	55862	1.516	1655 ms	1350 ms	1.226
<code>csma3_4</code>	1460287	2396727	1.641	770204 ms	53795 ms	14.317

Figure 20: Experimental results using the *csma* models, comparing our implementation against the sequential topological algorithm.

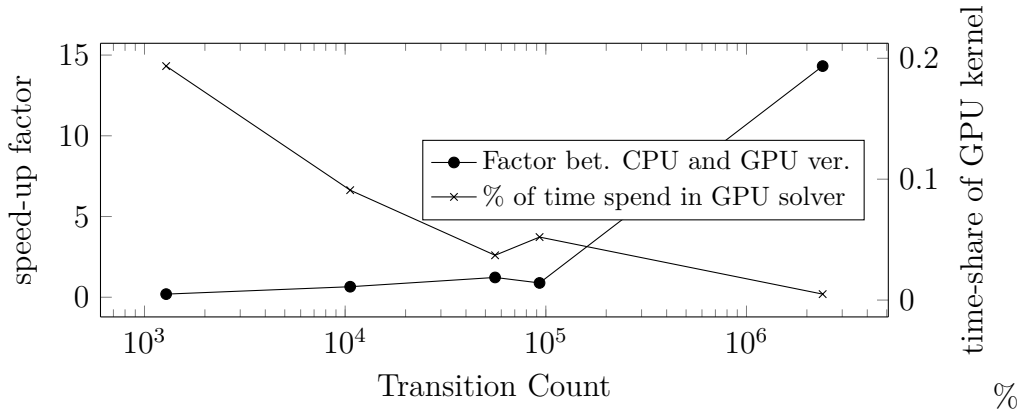


Figure 21: Speed-up factor against the sequential topological algorithm and time-share of the parallelized algorithm plotted against the transition count of the *csma* models.

example	states	transitions	$\frac{\text{transitions}}{\text{states}}$	time (CPU)	time (CUDA)	factor
csma2_2	1038	1282	1.235	35 ms	155 ms	0.226
csma2_4	7958	10594	1.331	210 ms	330 ms	0.636
csma2_6	66718	93072	1.395	1840 ms	2015 ms	0.913
csma3_2	36850	55862	1.516	1280 ms	1350 ms	0.948
csma3_4	1460287	2396727	1.641	54859 ms	53795 ms	1.020

Figure 22: Experimental results using the *csma* models, comparing our implementation against the StoRM standard algorithm.

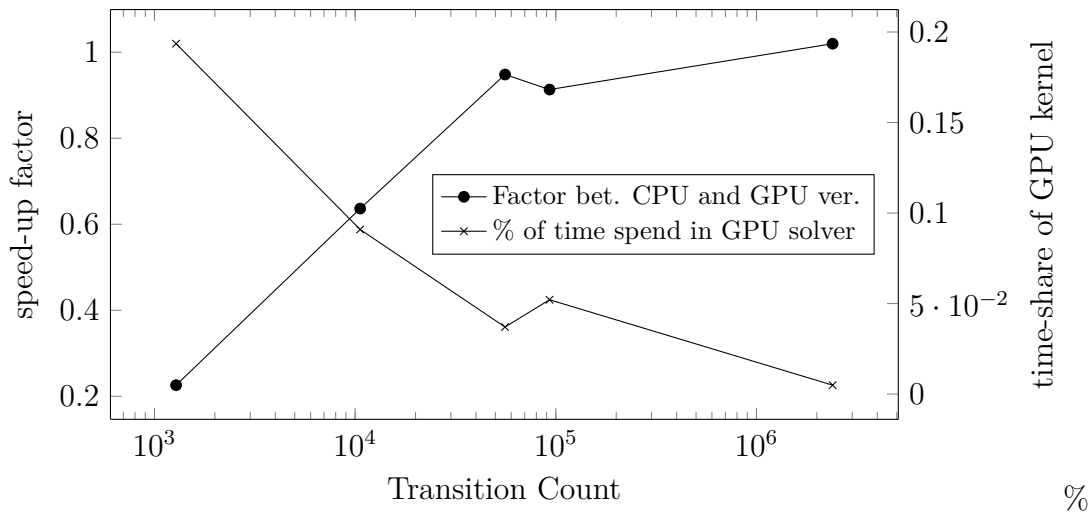


Figure 23: Speed-up factor against the StoRM standard model checker and time-share of the parallelized algorithm plotted against the transition count of the *csma* models.

Again, the measurement against the sequential topological algorithm results in an unusually high execution time and speed-up in the instance *csma3_4*. To rule out a bug in the StoRM framework, we will refer to the results in figure 22, which allow a more conservative comparison. In these instances the ratio between states and transitions is between ca. 1.5 to 2. Even though the biggest instance has more than $1.4 \cdot 10^6$ states, the speed-up is low when compared to the *leader* case study. We observe that a higher ratio between transitions and states allows for a better speed-up, possibly because of the one-warp-per-row/choice approach.

4.4. wlan and wlan_collide case study

Fourthly, we chose a case study by Kwiatkowska et al. [22] concerning the IEEE 802.11 Wireless LAN CSMA/CA (carrier sense, multiple access with collision avoidance) protocol. In contrast to the IEEE 802.3 CSMA/CD protocol the CA variant was designed with wireless communications in mind where situations can arise in which a sender can not observe another ongoing and overlapping transmission as, other than the receiver, the sender might not be in range of the second active sender. An exponential waiting scheme is used when detecting crosstalk.

ex.	states	transitions	$\frac{\text{transitions}}{\text{states}}$	time (CPU)	time (CUDA)	factor
wlan2	28480	57164	2.007	70 ms	200 ms	0.350
wlan3	96302	204576	2.124	178 ms	290 ms	0.612
wlan4	345000	762252	2.209	878 ms	915 ms	0.959
wlan5	1295218	2929960	2.262	2588 ms	2570 ms	1.007
wlan6	5007548	11475748	2.292	12725 ms	12275 ms	1.037

Figure 24: Experimental results using the *wlan* models, comparing our implementation against the sequential topological algorithm.

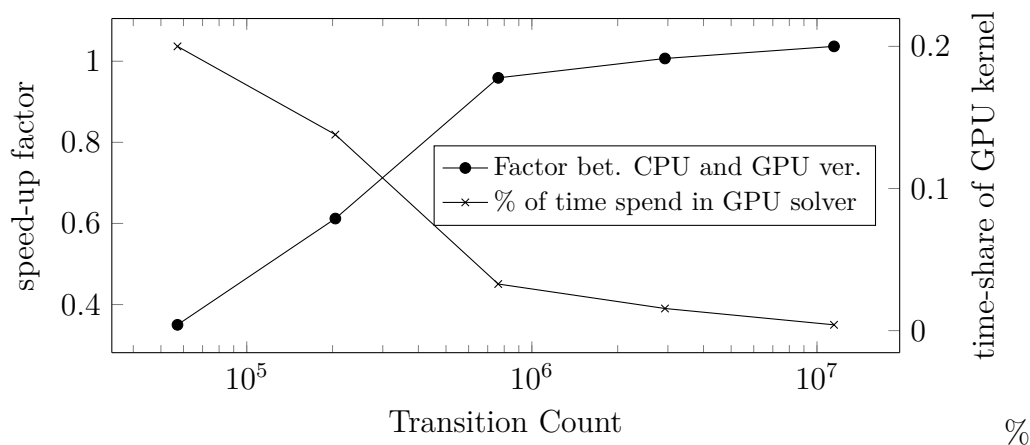


Figure 25: Speed-up factor against the sequential topological algorithm and time-share of the parallelized algorithm plotted against the transition count of the *wlan* models.

Results of the `wlan_collide` models:

ex.	states	transitions	$\frac{\text{transitions}}{\text{states}}$	time (CPU)	time (CUDA)	factor
0_2	6063	10619	1.751	15 ms	150 ms	0.100
0_4	11943	20965	1.755	35 ms	180 ms	0.194
0_6	17823	31311	1.757	60 ms	235 ms	0.255
1_2	10978	20475	1.865	20 ms	150 ms	0.133
1_4	26688	50533	1.893	75 ms	210 ms	0.357
1_6	42398	80591	1.901	150 ms	300 ms	0.500
2_2	28598	57332	2.005	55 ms	170 ms	0.324
2_4	59416	119957	2.019	150 ms	260 ms	0.577
2_6	107854	219439	2.035	355 ms	470 ms	0.755
3_2	96420	204744	2.123	185 ms	305 ms	0.607
3_4	118280	249381	2.108	265 ms	375 ms	0.707
3_6	284446	607711	2.136	835 ms	830 ms	1.006
4_2	345118	762420	2.209	685 ms	785 ms	0.873
4_4	345120	762422	2.209	715 ms	820 ms	0.872
4_6	728990	1605407	2.202	1790 ms	1730 ms	1.035
5_2	1295336	2930128	2.262	2605 ms	2710 ms	0.961
5_4	1295338	2930130	2.262	2645 ms	2800 ms	0.945
5_6	1591710	3563103	2.239	3575 ms	3485 ms	1.026
6_2	5007666	11475916	2.292	10505 ms	10515 ms	0.999
6_4	5007668	11475918	2.292	10535 ms	10560 ms	0.998
6_6	5007670	11475920	2.292	10635 ms	10695 ms	0.994

Figure 26: Experimental results using the `wlan_collide` models, comparing our implementation against the sequential topological algorithm.

The protocol is modeled in two separate models, one for checking successful message transfers and the other one for checking collision probabilities. The results from the sequential topological algorithm and the StoRM standard algorithm are very similar, we therefore only present those of the sequential version. The omitted data can be found in the Appendix.

Initially, our implementation performs worse than the sequential version, which is to be expected considering the overhead of invoking CUDA kernels, etc. From approximately 800,000 transitions upwards a comparable processing time is achieved. Major performance gains are opposed by the heavily decreasing percentage of time spent in our algorithm. With only 0.4% of the total execution time spent in the parallel algorithm the instance `wlan6` is a prime example for this problem.

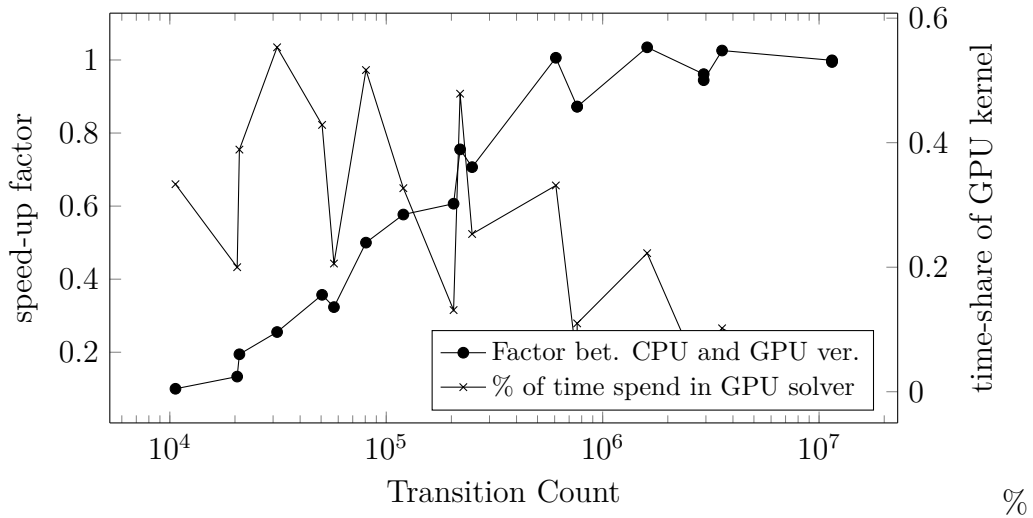


Figure 27: Speed-up factor against the sequential topological algorithm and time-share of the parallelized algorithm plotted against the transition count of the *wlan_collide* models.

5. Conclusion

In this thesis we presented a GPGPU aided parallel implementation of the value iteration algorithm, an iterative approach to compute minimal and maximal reachability properties in Markov decision processes by Bellman [4]. We observed that the initial overhead, i.e. copying data onto the GPGPU device, initializing resources, etc., of our method makes for worse performance in small examples than other, CPU-based implementations. On the other hand we pointed out a performance gain of more than a factor of three in examples that require a sizeable amount of iterations. The trend shown in all benchmarks alike is an indication that the preprocessing steps and other work done by the StoRM framework do not scale as well as our implementation does. Furthermore we found the decomposition into strongly connected components that we use as an option to solve models which do not fit into the GPU memory to be a major performance issue. For the models *coin8_2* and *leader8* nearly 20% of the total processing time was spent in the decomposition step. When analyzing the graphs given in chapter 4 we see that for examples like *csma* or *wlan*, the used model sizes were not large enough to see a speed-up. Larger instances require more than the 32 GB of RAM available on our benchmark workstation and could therefore not be evaluated. On

the other hand we found a possible correlation between achieved speed-up and the sparseness (or rather, denseness) of the generated transition probability matrices. From the graph depicted in figure 28 we observe that starting from around three times as many transitions as states our method begins to outperform the sequential CPU version. When comparing the possible peak performance of the GPGPU device with the achieved values we fall short of expectations. We explain this gap with several aspects. Firstly, the complexity of writing efficient CUDA kernels. A central issue is the even distribution of work onto threads. Neither too much nor too little work makes for a good throughput, and depending on the structure of the model, a different kernel design may work more efficiently. Secondly, a hardware deficiency concerning the usage of *double* precision floating point arithmetic. Recent consumer grade GPGPU devices take a significant performance penalty when executing operations on doubles instead of floats (see Ryoo et al. [28]) as there is only one arithmetic co-processor per warp. Thirdly, from the graphs presented we can see that the percentage spent in the code actually computing probabilities falls off fast to values as low as 0.04% in the *csma* models. This last observation may serve as a good explanation for our results and is one of the key issues to attract our interest in future work. We therefore conclude that while possible gains are promising, the use of GPUs is a difficult task that is no universal remedy for performance problems, but rather a promising addition to the model checking tools available, each tailored for certain fields of application.

5.1. Future Work

A next step towards a better GPU-aided model checking process would be the parallelization of the SCC decomposition since for the large model sizes for which our implementation gained significant speed-up, this preprocessing is necessary and takes a considerable amount of time.

Another important step would be the parallel implementation of the graph-based algorithms that determine which states belong to the sets S_0 and S_1 for a minimal or maximal reachability property. We discovered that with increasing model size, the share of these graph-algorithms in the total processing time grew to a substantial amount. Any significant progress can only be made after these algorithms have been successfully parallelized.

Following the interpretation of figure 28, a simple heuristic could use the factor between transition and state count to select either a sequential or parallel algorithm.

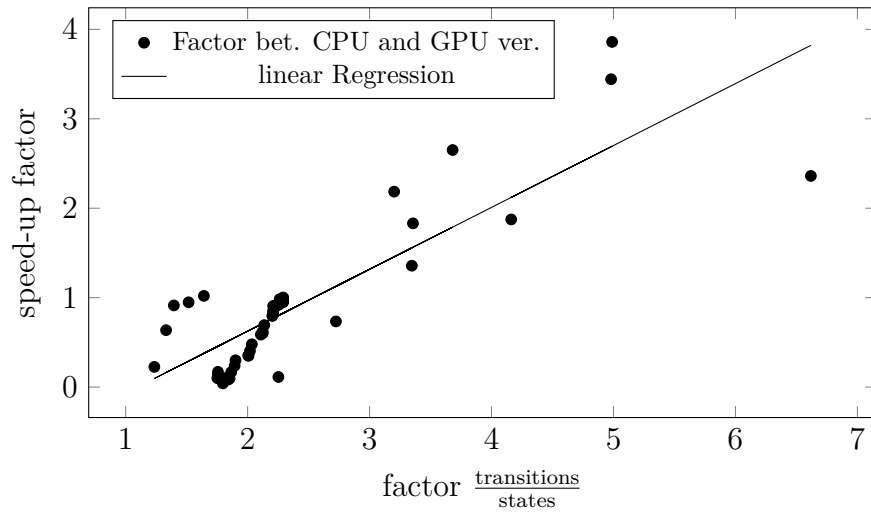


Figure 28: Comparing the sparseness of the input matrices against the achieved speed-up with respect to the StoRM standard algorithm.

A. Bibliography

References

- [1] Aspnes, J. and Herlihy, M. (1990). Fast randomized consensus using shared memory. *Journal of algorithms*, 11(3):441–461.
- [2] Baier, C., Katoen, J.-P., et al. (2008). *Principles of model checking*, volume 26202649. MIT press Cambridge.
- [3] Barnat, J., Bauch, P., Brim, L., and Češka, M. (2012). Designing fast ltl model checking algorithms for many-core gpus. *Journal of Parallel and Distributed Computing*, 72(9):1083–1097.
- [4] Bellman, R. (1957). A markovian decision process. Technical report, DTIC Document.
- [5] Bošnački, D., Edelkamp, S., and Sulewski, D. (2009). Efficient probabilistic model checking on general purpose graphics processors. In *Model checking software*, pages 32–49. Springer.
- [6] Bosnacki, D., Edelkamp, S., Sulewski, D., and Wijs, A. (2010). Gpu-prism: An extension of prism for general purpose graphics processing units. In *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*, pages 17–19. IEEE.
- [7] Bylina, B., Bylina, J., and Karwacki, M. (2011). Computational aspects of gpu-accelerated sparse matrix-vector multiplication for solving markov models. *Theoretical and Applied Informatics*, 23(2):127–145.
- [8] Cormie-Bowins, E. (2012). A comparison of sequential and gpu implementations of iterative methods to compute reachability probabilities. *arXiv preprint arXiv:1210.6412*.
- [9] D’Argenio, P. R., Jeannet, B., Jensen, H. E., and Larsen, K. G. (2001). Reachability analysis of probabilistic systems by successive refinements. In *Process Algebra and Probabilistic Methods. Performance Modelling and Verification*, pages 39–56. Springer.
- [10] Dijkstra, E. W. (1976). *A Discipline of Programming*, volume 4. Prentice-Hall Englewood Cliffs.

- [11] Fang, J., Varbanescu, A. L., and Sips, H. (2011). A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE.
- [12] Gabow, H. N. (2000). Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3–4):107–114.
- [13] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.
- [14] Hoberock, J. and Bell, N. (2010). Thrust: A parallel template library. Version 1.7.2.
- [15] Holzmann, G. J. and Bosnacki, D. (2007). The design of a multicore extension of the spin model checker. *Software Engineering, IEEE Transactions on*, 33(10):659–674.
- [16] Howard, R. A. (1960). Dynamic programming and markov processes.
- [17] Itai, A. and Rodeh, M. (1990). Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60–87.
- [18] Karimi, K., Dickson, N. G., and Hamze, F. (2010). A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*.
- [19] Katoen, J.-P., Zapreev, I. S., Hahn, E. M., Hermanns, H., and Jansen, D. N. (2011). The ins and outs of the probabilistic model checker {MRMC}. *Performance Evaluation*, 68(2):90 – 104. Advances in Quantitative Evaluation of Systems {QEST} 2009.
- [20] Knuth, D. E. and Yao, A. C. (1976). The complexity of nonuniform random number generation. *Algorithms and complexity: New Directions and Recent Results*, pages 357–428.
- [21] Kwiatkowska, M., Norman, G., and Parker, D. (2011). Prism 4.0: Verification of probabilistic real-time systems. In *Computer aided verification*, pages 585–591. Springer.
- [22] Kwiatkowska, M., Norman, G., and Sproston, J. (2002). *Probabilistic model checking of the IEEE 802.11 wireless local area network protocol*. Springer.
- [23] Kwiatkowska, M., Norman, G., Sproston, J., and Wang, F. (2004). Symbolic model checking for probabilistic timed automata. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 293–308. Springer.

- [24] Kwiatkowska, M., Norman, G., Sproston, J., and Wang, F. (2007). Symbolic model checking for probabilistic timed automata. *Information and Computation*, 205(7):1027–1077.
- [25] Moore, E. F. (1959). Shortest path through a maze. *Proceedings of the International Symposium on Theory of Switching*, pages 285–292.
- [26] Puterman, M. L. (1994). Markov decision processes. *John Wiley & Sons, New Jersey*.
- [27] Reiter, M. K. and Rubin, A. D. (1998). Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security (TISSEC)*, 1(1):66–92.
- [28] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-m. W. (2008). Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM.
- [29] Segala, R. (1996). Modeling and verification of randomized distributed real-time systems.
- [30] Shapley, L. S. (1953). Stochastic games. *Proceedings of the National Academy of Sciences of the United States of America*, 39(10):1095.
- [31] Sharir, M. (1981). A strong connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications*, 7(1):67–72.
- [32] Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160.
- [33] Tewarson, R. P. (1973). *Sparse matrices*. Academic Press, Inc.
- [34] Van Nunen, J. (1976). A set of successive approximation methods for discounted markovian decision problems. *Zeitschrift fuer Operations Research*, 20(5):203–208.
- [35] Wachter, B. (2011). *Refined probabilistic abstraction*. Logos Verlag Berlin GmbH.

B. Appendix

example	states	transitions	$\frac{transitions}{states}$	time (CPU)	time (CUDA)	factor
wlan2	28480	57164	2.007	70 ms	200 ms	0.350
wlan3	96302	204576	2.124	175 ms	290 ms	0.603
wlan4	345000	762252	2.209	830 ms	915 ms	0.907
wlan5	1295218	2929960	2.262	2525 ms	2570 ms	0.982
wlan6	5007548	11475748	2.292	12300 ms	12275 ms	1.002

Figure 29: Experimental results using the *wlan* models, comparing our implementation against the StoRM standard algorithm.

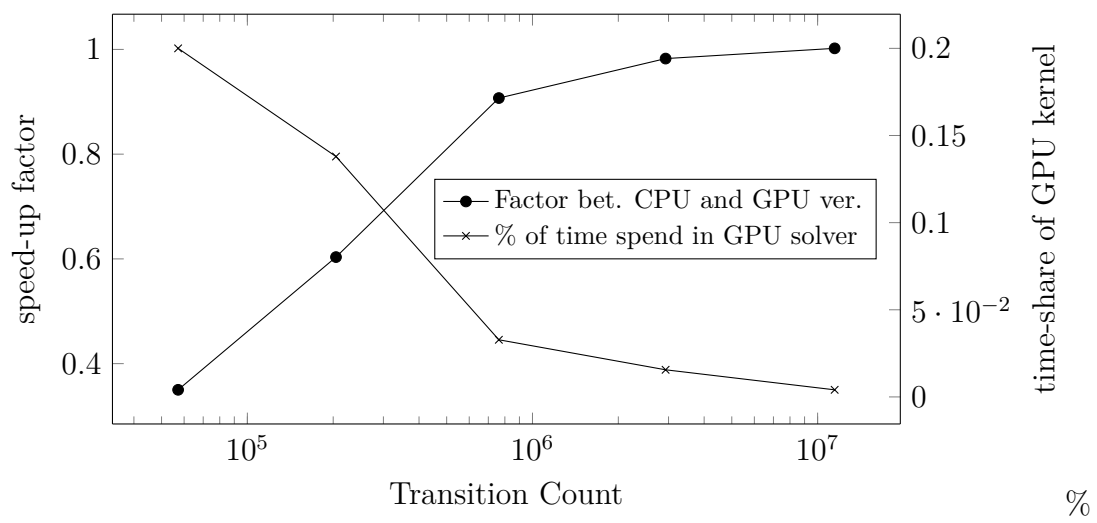


Figure 30: Speed-up factor against the StoRM standard model checker and time-share of the parallelized algorithm plotted against the transition count of the *wlan* models.

Results of the *wlan_collide* models:

ex.	states	transitions	$\frac{transitions}{states}$	time (CPU)	time (CUDA)	factor
0_2	6063	10619	1.751	15 ms	150 ms	0.100
0_4	11943	20965	1.755	25 ms	180 ms	0.139
0_6	17823	31311	1.757	40 ms	235 ms	0.170
1_2	10978	20475	1.865	25 ms	150 ms	0.167
1_4	26688	50533	1.893	50 ms	210 ms	0.238
1_6	42398	80591	1.901	90 ms	300 ms	0.300
2_2	28598	57332	2.005	60 ms	170 ms	0.353
2_4	59416	119957	2.019	105 ms	260 ms	0.404
2_6	107854	219439	2.035	225 ms	470 ms	0.479
3_2	96420	204744	2.123	190 ms	305 ms	0.623
3_4	118280	249381	2.108	220 ms	375 ms	0.587
3_6	284446	607711	2.136	575 ms	830 ms	0.693
4_2	345118	762420	2.209	670 ms	785 ms	0.854
4_4	345120	762422	2.209	680 ms	820 ms	0.829
4_6	728990	1605407	2.202	1375 ms	1730 ms	0.795
5_2	1295336	2930128	2.262	2575 ms	2710 ms	0.950
5_4	1295338	2930130	2.262	2595 ms	2800 ms	0.927
5_6	1591710	3563103	2.239	3165 ms	3485 ms	0.908
6_2	5007666	11475916	2.292	10405 ms	10515 ms	0.990
6_4	5007668	11475918	2.292	10365 ms	10560 ms	0.982
6_6	5007670	11475920	2.292	10155 ms	10695 ms	0.950

Figure 31: Experimental results using the *wlan_collide* models, comparing our implementation against the StoRM standard algorithm.

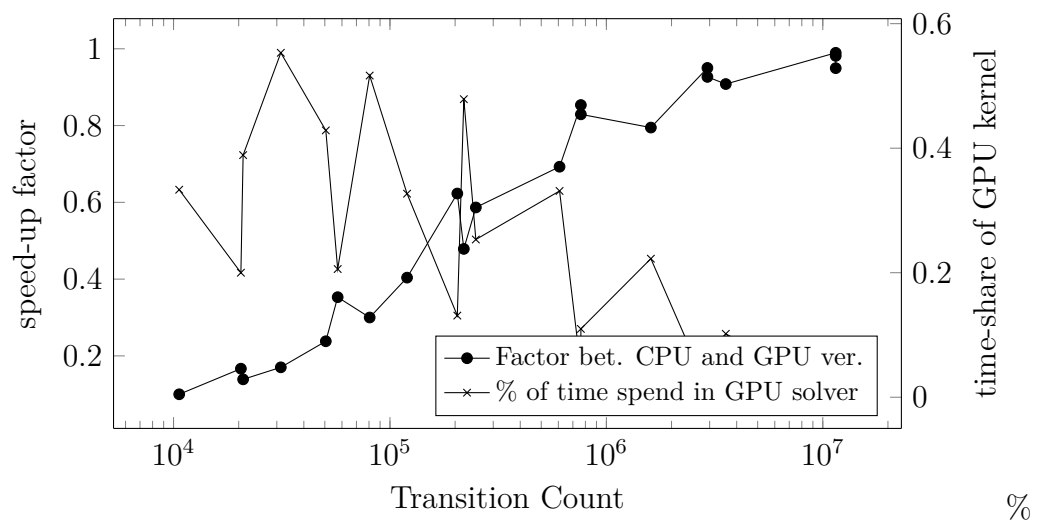


Figure 32: Speed-up factor against the StoRM standard model checker and time-share of the parallelized algorithm plotted against the transition count of the *wlan_collide* models.