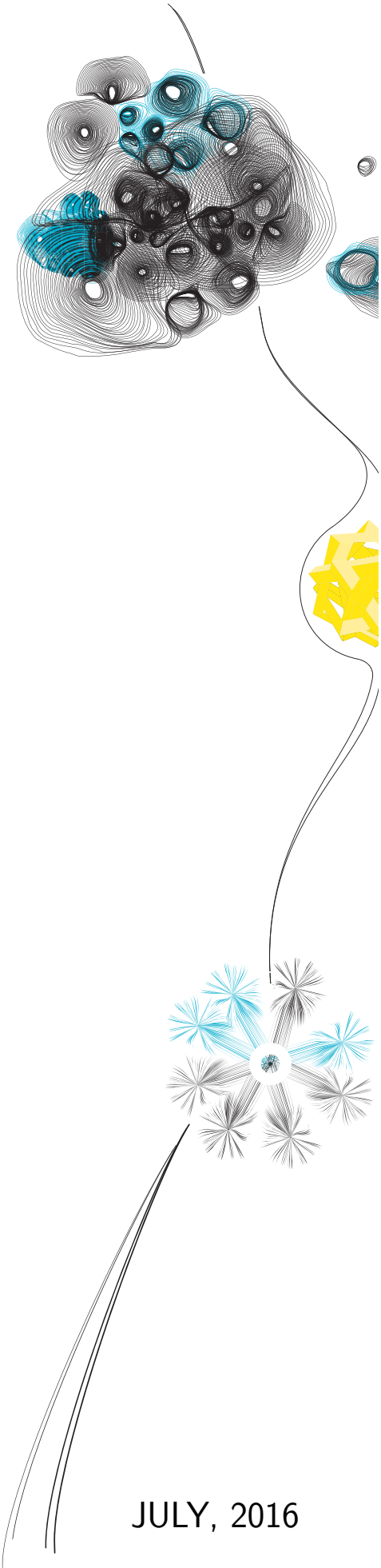


GPU IMPLEMENTATION OF PARTIAL-ORDER REDUCTION

MASTER THESIS

Thomas Neele

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Formal Methods and Tools

Exam committee:
Jaco van de Pol (UT), Stefan Blom (UT),
Anton Wijs (TU/e)

JULY, 2016

UNIVERSITY OF TWENTE.

Abstract

Model checking using GPUs has seen increased popularity over the last years. Because GPUs have only a limited amount of memory, only small to medium-sized systems can be verified. We improve memory efficiency for explicit-state GPU model checking by applying on-the-fly partial-order reduction. The correctness of the proposed algorithms is proved using a new version of the cycle proviso. Benchmarks show that our implementation achieves a reduction similar to or better than the state-of-the-art techniques for CPUs, while the runtime overhead is acceptable.

We also propose several optimisations for the tool GPUexplore. Benchmarks show that this results in a 7.8 times speed-up compared to the original implementation. For large models, our optimized version of GPUexplore can be more than two orders of magnitude faster than a sequential CPU implementation, reducing the runtime from more than an hour down to less than 37 seconds.

Contents

Chapter 1

Introduction

This chapter gives an introduction by providing a short overview of the field of model checking and general-purpose GPU programming. Then, it defines the contributions of this thesis and discusses related work. Finally, it gives an overview of the structure of this thesis.

1.1 Overview

In the research field of formal methods, model checking [2, 14] is a popular technique for proving the correctness of concurrent systems. However, the practical applicability is still limited by the problem of state-space explosion. This is due to the many possible interleavings of actions of concurrent processes and the many configurations of the state vector describing the process variables. In the early days, model checkers relied on the newest hardware to improve their performance. In recent years, however, sequential performance has not seen major improvements. A speed up can now only be gained by implementing parallel algorithms. While distributed systems saw a lot of popularity in the early 2000s, the focus has since shifted to the multi-core shared-memory architecture that is used in all modern-day hardware. Designing multi-threaded algorithms for this hardware brings forward new challenges: to achieve good scalability, resource contention between the threads should be kept to a minimum. Furthermore, subtle errors in the implementation may break the correctness. Most of the popular model checkers already have multi-core implementations [5, 13, 16, 23, 27].

Many-core architectures are a relatively new phenomenon and can mainly be found in graphics processing units (GPUs). Although GPUs are primarily aimed at rendering graphics to a screen, they can also be programmed for general tasks. This

is called general-purpose GPU (GPGPU) programming. Since the introduction of NVIDIA's CUDA [1] (a complete API for GPGPU programming) in 2008, GPUs have been used in many different applications, including model checking [4, 19, 44, 41].

Although GPUs do not seem suited to aid model checking due to their limited amount of memory, the massive number of threads that run in parallel makes GPUs attractive for this computationally intensive task. Their parallel power can speed-up state-space exploration by up to two orders of magnitude [3, 18, 41, 44]. Moreover, the amount of memory available on GPUs has increased significantly over the last years. Therefore, it is interesting to investigate the potential of GPUs, at least for future interest. Research has shown that, although GPUs can greatly outperform CPUs, they quickly run out of memory for larger models [7]. Therefore, the practical applicability of GPU model checking is still limited.

An approach to increase the memory efficiency of explicit-state model checking is by applying reduction techniques. Several approaches have been proposed, among others: *partial-order reduction* (POR) [38, 37, 22], *symmetry reduction* [25] and *bisimulation minimisation* [35]. All of these techniques exploit the fact that the state space may contain several states that are similar with respect to the property under consideration. By merging or not exploring the similar states, the memory footprint of the state-space exploration is reduced. When done efficiently, the amount of time required to check the property under consideration can also be reduced. Partial-order reduction and symmetry reduction can be performed on-the-fly, i.e. while exploring the state-space. Bisimulation minimisation can only be applied after the whole state space has been generated.

Contributions Our contributions are as follows:

1. We improve the memory efficiency of GPU model checking. To that end, we extend GPUexplore [41, 42], one of the first tools that runs a complete model checking algorithm on the GPU, with POR. We propose GPU algorithms for three practical approaches to POR, based on ample [24], cample [9] and stubborn sets [38].

2. We improve the cample-set approach by identifying clusters of processes on-the-fly. This removes the need to manually specify the structure of the input model and improves the reduction potential of the cample-set approach for many types of models.

3. We introduce a new, weaker version of the cycle proviso and use it to prove that our algorithms do not allow ignoring of actions. Furthermore, we also prove that our algorithms adhere to the other POR conditions.

4. We compare the performance of our POR algorithms on a GPU with LTSmin [27], a model checker that implements state-of-the-art algorithms for multi-core POR.

5. We propose several general optimizations to improve GPUexplore's runtime. We compare the performance of our implementation and the original version from [42] to determine the speed-up that follows from our optimizations. Additionally, we provide a comparison with CADP [20], a sequential CPU model checker.

The results related to POR from this thesis have also been published in [36].

1.2 Related Work

Partial-order reduction In multi-core model checking, there are several works on partial-order reduction: Barnat et al. [6] propose a new cycle proviso that is based on a topological sorting. A state-space cannot be topologically sorted if it contains cycles. This information is used to determine which states need to be fully expanded. Their implementation can obtain competitive reductions. However, it is not clear from the paper whether it is slower of faster than a standard DFS-based implementation.

Laarman and Wijs [32] designed a multi-core version of POR that is based on Laarman et al.'s earlier work on multi-core nested depth-first search [29]. Their algorithm yields better reductions than SPIN's implementation, but has higher runtimes. The scalability of the algorithm is good up to 64 cores.

Bošnački et al. have defined cycle provisos for breadth-first search [11] and general state expanding algorithms (GSEA) [12], a generalization of search algorithms, including depth-first and breadth-first search. They implemented this in an extension of SPIN. The results of the benchmarks show a significant improvement over the standard implementation of SPIN. Comparisons with other tools are not provided. Although the included algorithms are not multi-core, the theory is relevant for our design, since we will focus on GSEA.

GPU model checking GPGPU techniques have already been applied in model checking by several others, all with a different approach: DIVINE performs state-space generation on the CPU, but offloads the detection of cycles to the GPU [4, 3]. The GPU then applies the *Maximal Accepting Predecessors* (MAP) or *One Way*

Catch Them Young (OWCTY) algorithm to find these cycles. This results in a speed-up over both multi-core DIVINE and multi-core LTSmin.

Edelkamp and Sulewski [19] perform successor generation on the GPU and apply delayed duplicate detection to store the generated states in main memory. They implemented the ideas in a tool called CuDMoC. It was benchmarked running natively on a GPU as well as being emulated by a CPU. This comparison yields a significant speed-up of about 8 times of the GPU over the CPU emulation. CuDMoC performs better than DIVINE, it is faster and consumes less memory per state. The performance is worse than multi-core SPIN, although it should be noted SPIN was not always able to explore the complete state space.

GPUexplore by Wijs and Bošnački [41, 42] performs the complete model checking process on the GPU, including successor generation and storage of states. In addition, this tool can check for absence of deadlocks and can also check safety properties. The performance of GPUexplore is similar to LTSmin running on about 10 threads.

Bartocci et al. [7] have implemented a CUDA version of SPIN. Their approach is similar to GPUexplore: the state-space generation is completely done on the GPU. The implementation has a significant overhead for smaller models, but performs reasonably well for medium-sized state spaces.

Wu et al. [43] extend the model checker PAT with a CUDA implementation of counter-example generation. Their ideas can be applied to generate shortest counter-examples for SCC-based LTL model checking. After the CPU has generated the state space and performed SCC decomposition, the GPU explores small parts of the state space to compute the shortest path to an error state. The implementation applies dynamic parallelism to cope with the variable width of breadth-first search layers. The paper does not contain a performance comparison with other tools.

Wu et al. [44] also implemented a complete model checker in CUDA. They adopted several techniques from GPUexplore, and added mechanisms for dynamic parallelism and parallel generation of synchronizing transitions. The speed up gained from dynamic parallelism proved to be minimal. A comparison with traditional sequential state-space exploration shows a good speed-up, but it is not clear from the paper how the performance compares with other multi-core or many-core tools.

GPUs have also been applied in probabilistic model checking: Bošnački et al. [10, 40] speed up value-iteration for probabilistic properties by solving linear equation systems on the GPU. Češka et al [13] implemented parameter synthesis for parametrized continuous time Markov chains.

1.3 Thesis Structure

The rest of this report is structured as follows: Chapter 2 explains the theory required to read the other chapters. Then, Chapter 3 gives a detailed description of optimizations to speed-up state-space exploration. Chapter 4 describes the GPU POR algorithms we propose and formally proves why the proposed algorithms are correct. Next, Chapter 5 provides the results of experiments we performed. Finally, Chapter 6 gives a conclusion and several areas that require further study.

Chapter 2

Background

This chapter starts by introducing the basic theory of *transition systems* and *concurrent processes*. Then, it explains the ideas behind partial-order reduction. Finally, an introduction to the architecture of GPUs is given.

2.1 Transition Systems

Definition 1. A labelled transition system (LTS) is a tuple $\mathcal{T} = (S, A, \tau, \hat{s})$, where:

- S is the set of states.
- A is the set of actions.
- $\tau \subseteq S \times A \times S$ is the relation that defines transitions between states. Each transition is labelled with an action.
- $\hat{s} \in S$ is the initial state.

Let $\mathit{enabled}(s) = \{\alpha | (s, \alpha, t) \in \tau\}$ be the set of actions that is enabled in state s and $\mathit{succ}(s, \alpha) = \{t | (s, \alpha, t) \in \tau\}$ the set of successors reachable through some action α. Additionally, we lift *succ* to take a set of states or actions as argument respectively. The second argument of *succ* is omitted when all actions are considered: $\mathit{succ}(s) = \mathit{succ}(s, A)$. If $(s, \alpha, t) \in \tau$, then we write $s \xrightarrow{\alpha} t$. We call a sequence of actions and states $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n$ an *execution*. When the actions are left out of an execution, we call this a *path*: $\pi = s_0 \ldots s_n$. The sequence of actions observed along an execution is called an *action sequence*: $\alpha_1 \ldots \alpha_n$. If there exists a path $s_0 \ldots s_n$ such that $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n$, then we say that s_n is reachable from s_0. The set of all reachable states from a state s is the reflexive transitive closure of *succ*. The set of reachable states of an LTS $\mathcal{T}$ is equal to the set of states reachable from the initial state.

To specify concurrent systems consisting of a finite number of finite-state processes, we define a *network of LTSs* based on [33]. In this context we also refer to the participating LTSs as *concurrent processes*.

Definition 2. A network of LTSs is a tuple $\mathcal{N} = (\Pi, V)$, where:

- Π is a list of n processes $\Pi[1], \ldots, \Pi[n]$ that are modelled as LTSs.
- V is a list of synchronization rules. A synchronization rule is a tuple $(\vec{t}, a)$, where a is an action and $\vec{t} \in \{0, 1\}^n$ is a synchronization vector that denotes which processes take part in the synchronization on a.

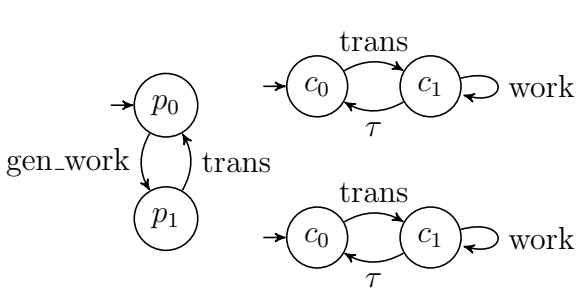
Based on the definition of a network of LTSs, we can distinguish two types of actions: (1) *local actions* that are not part of any synchronization rule and therefore cannot be blocked and (2) *synchronizing actions* that are part of at least one synchronization rule. Syncing actions are blocked when there is no applicable synchronization rule for that action. More formally: action α is blocked in state s when $\neg\exists(\vec{t}, a) \in V : (a = \alpha \wedge \forall i \in \{1 \ldots n\} : \vec{t}[i] = 1 \Rightarrow a \in \mathit{enabled}_i(s))$. Note that although processes can only synchronize on actions with the same name, this does not limit the expressiveness. Any network following a more general definition can be transformed into a network that follows our definition by proper action renaming.

Example 1. An example of an LTS network can be found in Figure 2.1. This network contains one producer and two consumers. After the producer generates work (action *gen_work*), the work is sent to one of the two consumers by synchronizing on the *send* action. The consumer that received the work, processes it (action *work*), until at some point it is ready to receive new work. The synchronization rules are specified using the EXP syntax on the right. The two synchronization rules represent the transmission of work to the first and second consumer respectively. An underscore indicates that a process does not synchronize based on that rule.

For every network, we can define an LTS that represents its state space.

Definition 3. Let $\mathcal{N} = (\Pi, V)$ be a network of processes. $\mathcal{T}_{\mathcal{N}} = (S, A, \tau, \hat{s})$ is the LTS induced by this network, where:

- $S = S[1] \times \cdots \times S[n]$ is the cross-product of all the state spaces.
- $A = A[1] \cup \cdots \cup A[n]$ is the union of all actions sets.
- $\tau = \{(\langle s_1, \ldots, s_n \rangle, a, \langle s'_1, \ldots, s'_n \rangle) | \exists(\vec{t}, a) \in V. \forall i \in \{1, \ldots, n\}. \vec{t}(i) = 1 \Rightarrow (s_i, a, s'_i) \in \tau_i \wedge \vec{t}(i) = 0 \Rightarrow s_i = s'_i\} \cup \{(\langle s_1, \ldots, s_n \rangle, a, \langle s'_1, \ldots, s'_n \rangle) | \exists i \in \{1, \ldots, n\}. (s_i, a, s'_i) \in \tau_i \wedge \forall j \neq i : s_i = s'_i\}$ is the transition relation that follows from each of the processes and the synchronization rules.




```
par using
    trans * trans *   _   -> trans,
    trans *   _   * trans -> trans
in
    "producer.aut"
    ||
    "consumer.aut"
    ||
    "consumer.aut"
end par
```

Figure 2.1: Example of LTS network with one producer and two consumers.

- $\hat{s} = \langle \hat{s}[0], \ldots, \hat{s}[n] \rangle$ is the initial state, which is a combination of the initial states of the processes.

The states of $\mathcal{T}_\mathcal{N}$ are vectors with n *slots*. The ith slot in a state s is called $s[i]$. We refer to each of the fields of process $\Pi[i]$ with $S[i]$, $A[i]$, $\tau[i]$ and $\hat{s}[i]$ respectively. The actions of process $\Pi[i]$ that are enabled in state s are referred to as $enabled_i(s) = enabled(s[i])$.

2.2 Exploration

Since the set of reachable states of a process network is restricted by the synchronization rules in most cases, it is hard to predict whether it contains some error state or other undesired behaviour. To decide this problem, the whole set of reachable states has to be constructed on a state-by-state basis, starting with the initial state. This is computationally a hard problem, due to the fact that the size of the state space grows exponentially with the amount of processes. This phenomenon is called *state-space explosion*.

A detailed procedure for state-space exploration is listed in Algorithm 1. States are stored in two sets: all the states that still need to be explored are in *Open* and all the states for which exploration has at least started are in *Closed*. On lines 2 and 3, one state s is selected from *Open* and moved to *Closed*. Then, all the successors of s that have not been explored yet are added to *Open* (line 6). The algorithm terminates when there are no states left to explore.

The implementation of *Open* influences the order in which states are visited. When *Open* is implemented as a stack, the exploration follows a depth-first search (DFS) order. When *Open* is implemented as a queue, the exploration follows a breadth-first search (BFS) order.

Algorithm 1: State-space exploration algorithm

```
   Data: Open ← {ŝ}, Closed ← ∅
1  while Open ≠ ∅ do
2      Open ← Open \ {s} for some s ∈ Open;
3      Closed ← Closed ∪ {s};
4      foreach t ∈ succ(s) do
5          if t ∉ Closed then
6              Open ← Open ∪ {t};
```

2.3 Partial-Order Reduction

To combat the state-space explosion, several reduction techniques have been proposed [38, 26, 35]. The general concept of reductions can be formally defined using a *reduction function*.

Definition 4. A reduced LTS can be defined according to some reduction function $r : S \to 2^A$. The reduction of $\mathcal{T}$ according to r is denoted by $\mathcal{T}_r = (S_r, A, \tau_r, \hat{s})$, such that:

- $(s, \alpha, t) \in \tau_r$ if and only if $(s, \alpha, t) \in \tau$ and $\alpha \in r(s)$.
- S_r is the set of states reachable from $\hat{s}$ under τ_r.

As we can see, for each reduction function r and transition system $\mathcal{T}$, there is a unique reduced system $\mathcal{T}_r$. Clearly, $\mathcal{T}_r$ does not depend on the way that r was computed. Although it is possible to compute the reduction function after generating the whole state space, this does not solve the problem of state-space explosion. We would likely run into memory limitations before the state-space generation even completed. Another option is to compute some restrictions on r beforehand [28]. However, this technique may offer significantly less reduction [34]. Therefore, the preferred solution is to compute the reduction function while generating the state space (*on-the-fly*). In that case, we decide which actions to preserve and which actions to prune at the moment a state is explored. The states to which the pruned actions lead are not explored at that time (see Figure 2.2). However, these states may be explored later if they are reachable through other transitions. The downside of performing some reduction algorithm on-the-fly is that there is no overview of the whole state space. Therefore, the choice for $r(s)$ is usually not optimal.

The main idea behind *partial-order reduction* is that not all interleavings of actions of the parallel processes are relevant to the property under consideration. It

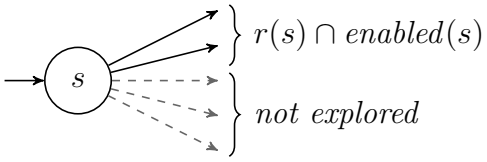


Figure 2.2: Transitions that are not defined under the reduction function r are not explored.

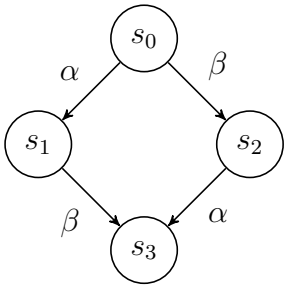


Figure 2.3: LTS with two independent actions α and β.

suffices to check only some of those interleavings. To reason about this, we define when actions are *independent*.

Definition 5. Two actions α, β, with $\alpha \neq \beta$ are independent in state s if and only if the following conditions hold:

- if $\beta \in enabled(s)$, then $\alpha \in enabled(s) \Leftrightarrow \alpha \in enabled(succ(s, \beta))$
- if $\alpha \in enabled(s)$, then $\beta \in enabled(s) \Leftrightarrow \beta \in enabled(succ(s, \alpha))$
- $succ(succ(s, \alpha), \beta) = succ(succ(s, \beta), \alpha)$

Actions that are not independent are called *dependent*.

Example 2. We consider a system where one process reads a variable x (action α) and another process writes variable y (action β). These actions are globally independent, because the order in which they are executed does not influence the result. This can be represented visually as in Figure 2.3. We call this a *diamond* structure.

From any execution in the system, we can obtain other interleavings by repeatedly permuting adjacent independent actions.

Example 3. Let $\alpha\beta_1\beta_2$ be an action sequence in $\mathcal{T}$, where α is globally independent of β_1 and β_2. Then $\beta_1\alpha\beta_2$ and $\beta_1\beta_2\alpha$ are also action sequences in $\mathcal{T}$. On the other hand, $\alpha\beta_2\beta_1$ is not an action sequence in $\mathcal{T}$, because β_1 and β_2 are dependent.

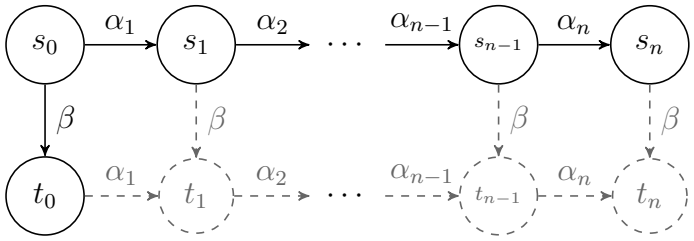


Figure 2.4: By repeatedly applying the persistence condition (C1) to the path $s_0 \dots s_i$, where $0 < i \leq n$, we can conclude that states $t_1 \dots t_n$ exist. Therefore, C1 enforces that we can only choose $r(s_0) = \{\beta\}$ when β is independent of all α_i for $0 < i \leq n$. In this way, the persistence condition is prevents loss of behaviour.

2.3.1 Persistent Sets

Based on the theory of independent actions and their interleavings, the following restrictions on the reduction function have been defined [12, 37]:

C0a $r(s) \subseteq \mathit{enabled}(s)$.

C0b $r(s) = \emptyset \Leftrightarrow \mathit{enabled}(s) = \emptyset$.

C1 For all $s \in S$ and executions $s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_{n-1}} s_{n-1} \xrightarrow{\alpha_n} s_n$ such that $\alpha_1, \dots, \alpha_n \notin r(s)$, α_n is independent in s_{n-1} with all actions in $r(s)$.

C0a and C0b make sure that the reduction does not introduce new behaviour and new deadlocks respectively. C1 implies that all $\alpha \in r(s)$ are independent of $\mathit{enabled}(s) \setminus r(s)$. Informally, this means that only the execution of independent actions can be postponed to a later state (cf. Figure 2.4). A set of actions that satisfies these criteria is called a *persistent set*. It is hard to compute the smallest persistent set, therefore several practical approaches have been proposed, which will be introduced in Chapter 4.

Any persistent set preserves deadlocks and can therefore be used to check a system for deadlocks. However, we are also interested in *safety* properties, which are generally not preserved. Therefore, we have to address another issue: the *action-ignoring problem*.

2.3.2 Action Ignoring

When exploring a state space and applying partial-order reduction, it is possible that the execution of a certain action is postponed indefinitely, i.e. there is an action

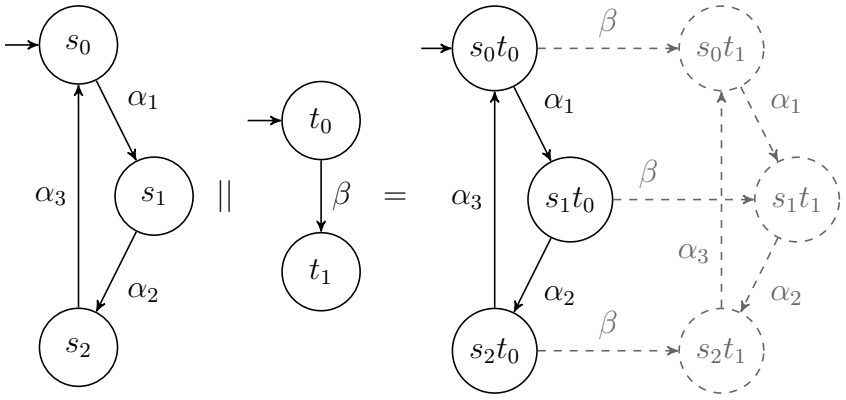


Figure 2.5: Indefinite ignoring of action β.

that is not part of $r(s)$ for any state $s \in \mathcal{T}_r$. Because we are dealing with finite state-spaces and we have to satisfy condition C0b, this can only happen in a cycle.

Example 4. In Figure 2.5, we see the parallel composition of two processes. In the parallel composition, action β is ignored indefinitely along one of the loops, which leads to states s_0t_0, s_1t_1 and s_2t_1 never being explored.

In order to preserve safety properties, we need to impose another condition on the reduction function. This condition is called the *action ignoring proviso* and it prevents actions from being postponed for ever.

C2ai For every state $s \in S_r$ and every action $\alpha \in \mathit{enabled}(s)$, there exists an execution $s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n$ in the reduced state space, such that $\alpha \in r(s_n)$.

By preventing ignoring of actions, any transition label that occurs in the original state space, also occurs in the reduced state space [38]. Therefore, violations of safety properties (indicated by a special error transition either in one of the processes or in an additional *monitor* process) are also preserved. Note that other temporal properties are not preserved, but they fall outside of the scope of this thesis.

The problem with the action ignoring proviso is that it requires an overview of the reduced state-space. Moreover, the problem of finding the minimal amount of states covering all cycles is NP-complete. Therefore, it is not possible to apply these provisos directly in an on-the-fly POR algorithm.

Provisos for Safety Properties

We can define a stronger version of this proviso that can be decided on a per-state basis. Then, the information about the search history can be used to decide whether

it is possible to further postpone the execution of some action. As an example, we consider POR under depth-first search. When a certain state s is expanded and we find a transition leading back to a state that is on the DFS stack, we have found a cycle. In case all actions in $r(s)$ lead back to the stack, this might result in action-ignoring. In that case we should either try to find another set of actions for our reduction or fully expand s. This strategy is formally defined in condition C2s (C2-stack):

C2s There is at least one action $\alpha \in r(s)$ and state t such that $s \xrightarrow{\alpha} t$ and t is not in the DFS stack. Otherwise, $r(s) = \mathit{enabled}(s)$.

Condition C2s implies C2ai [11], therefore C2s prevents the ignoring of actions. For algorithms that do not follow a DFS order, the following, more general *closed-set proviso* [12] can be used as an alternative for C2s.

C2c There is at least one action $\alpha \in r(s)$ and state t such that $s \xrightarrow{\alpha} t$ and $t \notin \mathit{Closed}$. Otherwise, $r(s) = \mathit{enabled}(s)$.

2.4 GPU Architecture

CUDA is a programming interface developed by NVIDIA to enable general purpose programming on a GPU [1]. It provides a unified view of the GPU ('device'), simplifying the process of developing for multiple devices. Code to be run on the device ('kernel') can be programmed using a subset of C++. The kernel specifies the behaviour of a single thread. When the kernel is executed, multiple threads execute the same code in parallel.

Considering the hardware, a GPU is divided up into several *streaming multiprocessors* (SM) that each contain a large amount of cores. On the side of the programmer, threads are grouped into blocks. When assigning work to a GPU, the number of threads per block and the number of blocks need to be specified. The GPU then schedules the thread blocks on the streaming multiprocessors. One SM can run multiple blocks at the same time, but one block cannot execute on more than one SM. The SM manages threads in groups of 32 threads, called *warps*. Threads in a warp execute instructions in lock-step fashion. Figure 2.6 provides an overview of the software and hardware thread hierarchy.

Another important aspect of the GPU architecture is the memory hierarchy. Firstly, each SM has *shared memory* that is divided between the blocks running on that SM. The shared memory assigned to a block can only be accessed by the threads

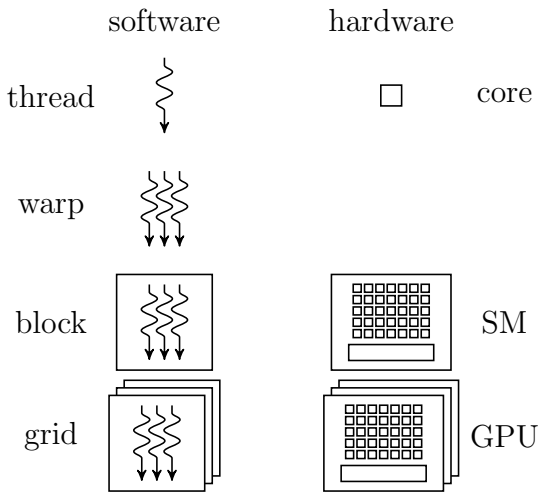


Figure 2.6: Summary of software and hardware thread hierarchy in CUDA.

in that block. The shared memory is placed on-chip, therefore it has a low latency. Secondly, there is the global memory that can be accessed by all the threads. It has a high bandwidth, but also a high latency. The amount of global memory is typically multiple gigabytes. There are three caches for the global memory: the L1 cache, the L2 cache and the texture cache. Data in the global memory that is marked as read-only (a 'texture') may be placed in the texture cache. The global memory can also be accessed by the CPU ('host'), thus it serves as an interface between the host and the device. Figure 2.7 provides a schematic overview of the architecture.

The key to writing efficient GPGPU programs is making optimal use of the architecture. The work should be divided into small tasks that will be executed by the thread blocks. All threads in a block work together on this small task. The work division strategy plays an important role, because communication between blocks is only possible via global memory. Therefore, data dependencies between blocks should be avoided.

Warps influence the performance of GPU algorithms in several ways. Firstly, the amount of branch divergence within warps should be kept to a minimum. Consider an if-statement with condition C, where C is *true* for at least one thread and *false* for at least one thread. Since the threads in one warp step through all instructions together, the two branches will be executed sequentially. This can lead to a reduction in performance.

On the other hand, the concept of warps can be exploited when fetching data from memory. When the threads in a warp fetch a continuous block of 32 integers

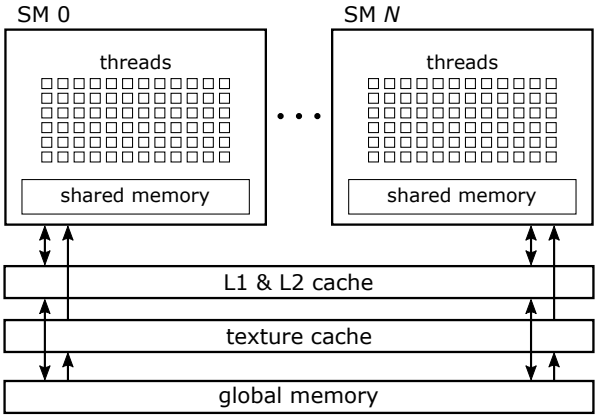


Figure 2.7: Schematic overview of the GPU architecture.

from global memory, this memory access can be executed in parallel. This is called *coalesced access*. Additionally, this data fits exactly in one cache line of 128 bytes.

Finally, there are several instructions for quickly exchanging data between threads of a warp. These *warp instructions* are implemented by register swapping. CUDA's memory model does not guarantee visibility of writes without a barrier synchronization, even between threads of the same warp. Therefore, it is often faster to apply warp instructions and avoid synchronization of the whole block.

Example 5. Figure 2.8 illustrates how warp instructions enable fast data sharing. Here, we will compute the minimum value of a variable a of all threads in a warp using *butterfly reduction*. For this reduction algorithm, the `__shuffle` instruction is used. This function has two arguments: the first argument is the value to exchange and the second argument is the source *lane* (index of a thread in its warp). It returns the value of the first argument as computed by the source lane. For this algorithm, we compute the source lane by applying an offset. After each shuffle operation, the minimum value of variable a is recomputed and the offset is increased by a factor two. In this way, we achieve a tree-like structure for the data exchanges and only five instructions are necessary to compute a minimum from the variables of 32 threads. Although Figure 2.8 only displays the information flow relevant for the first thread, the same algorithm is executed by all threads in a warp. Therefore, when the algorithm terminates, all 32 threads have computed the same minimum value.

The use of shared memory can be an important factor in the performance of GPGPU applications. However, there is only a limited amount of shared memory available per SM, typically no more than several tens of kilobytes. Therefore, the

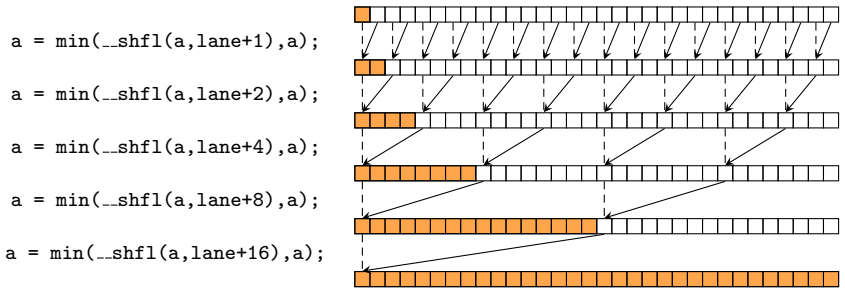


Figure 2.8: Computing the minimum of variable a over all threads of one warp with butterfly reduction. The flow of data relevant for the first thread of the warp ($lane = 0$) is indicated by the arrows. Of course, variable a also carries over between iterations, as indicated by dashed lines. The value of variable a for the first thread is the minimum of all as of the orange-coloured threads after each step.

data that is accessed most often should be stored in shared memory. In that way, it serves as a user-managed cache for global memory. Storing critical data in shared memory helps to reduce the amount of memory accesses to the global memory.

Chapter 3

Optimizations

This chapter first introduces the existing architecture of GPUexplore. Then, it discusses how we optimized the implementation to minimize the runtime required for state-space exploration.

3.1 Existing Design

GPUexplore is a model checker that can check for deadlocks and safety properties. GPUexplore executes all the computations on the GPU and does not rely on any processing by the CPU. The main kernel of GPUexplore implements lines 2-6 of Algorithm 1. This kernel is launched repetitively until all states have been explored.

The *Open* and *Closed* set are implemented by a single hash table that occupies most of the global memory. All states are stored in this hash table. Whether a state is new (in the *Open* set) or old (in the *Closed* set) is indicated by a single bit in the state vector. This bit is not considered by the hash function. This ensures that a state will be inserted in the same position, regardless of the value of the new/old bit.

The hash table uses open addressing with rehashing. Since we are not interested in deleting states from the hash table, it only needs to support a *findOrPut* operation. This operation tries to find an element, and if it is not present, the element is inserted. The implementation of findOrPut is thread-safe: it does not allow for data races. In addition, it is lock-less: the atomic *compareAndSet* (CAS) operation is used to guarantee thread-safety. The hash table is divided into buckets of 32 integers. Each bucket may contain multiple state vectors. Whenever possible, threads cooperate with the other threads in their warp and read/write a complete bucket to ensure coalesced access.

The threads are organized as follows: each thread is primarily part of a block.

As described in section 2.4, the hardware enforces that threads are grouped in warps of size 32. We also created logical groups, called *vector groups* (note that vector groups are not a CUDA concept). The number of threads in a vector group is equal to the number of processes in the network (cf. section 2).The threads in a vector group cooperate to compute the successors of a single state. Each thread has a vector group thread index (*vgtid*) and is responsible for generating the successors of process $\Pi[vgtid]$. Successors following from synchronizing actions are generated in cooperation. Threads with vgtid 0 are *group leaders*. Note that the algorithms presented here specify the behaviour of one thread, but are run on multiple threads and on multiple blocks. Most of the synchronization is hidden in the functions that access shared or global memory.

Algorithm 2: GPUexplore exploration framework

```
   Data: __global__ table[ ]
   Data: __shared__ workTile[ ], cache[ ]
1  vgid ← tid / numProc;                          /* index of the vector group */
2  vgtid ← tid mod numProc;                       /* id of the thread in the group */
3  foreach i ∈ 0 ... NumIterations do
4      workTile ← gatherWork();
5      __syncthreads();
6      s ← workTile[vgid];
7      foreach t ∈ succ_vgtid(s) do
8          storeInCache(t);
9      __syncthreads();
10     foreach t ∈ cache do
11         if isNew(t) then
12             findOrPutWarp(t);
13             markOld(t);
```

A high-level view on the algorithm of GPUexplore is presented in Algorithm 2. This kernel is executed repetitively until all reachable states have been explored. In between two kernel launches, the CPU only queries the GPU to determine whether exploration has finished. Several *kernel iterations* may be performed during each launch of the kernel (NumIterations is set by the user). Each iteration starts with *work gathering*: blocks search for unexplored states in global memory and copy those states to the work tile in shared memory (line 4). Once the work tile is full, the `__syncthreads` function from the CUDA API synchronizes all threads in the block and guarantees that writes to the work tile are visible to all threads (line 5). Then, each vector group takes a state from the work tile (line 6) and generates its successors (line 7). To prevent non-coalesced accesses to global memory, the successors are first

placed in a cache in shared memory (line 8). When all the vector groups in a block are done with successor generation, each warp scans the cache for new states and copies them to global memory (line 12). The states are then marked old in the cache (line 13), so they are still available for local duplicate detection later on. For details on how successors are computed and the inner workings of the hash table, we refer to [42].

3.2 Improvement of Work Scanning

At the beginning of each iteration, each block fills its work tile by linearly scanning the hash table in global memory (cf. Algorithm 3). Each warp linearly scans its section of the hash table by stepping a *nrOfWarps* amount of buckets in each iteration (line 14). When a new state is found (line 9), the amount of states in the work tile (stored in *tileCount*) is incremented atomically (line 10). If the previous value of *tileCount* (stored in j) is smaller than the size of the work tile (line 11), then the state is copied from the hash table into the work tile and marked as old in the hash table (lines 12-13).

Algorithm 3: Work gathering

```
   Data: __global__ table[ ]
   Data: __shared__ workTile[ ], tileCount
 1 function gatherWork():
 2     lane ← threadId mod warpSize;
 3     warpId ← threadId/warpSize;
 4     globalWarpId ← (nrOfBlocks/warpSize) * blockId + warpId;
 5     nrOfWarps ← (blockSize/warpSize) * nrOfBlocks;
 6     i ← globalWarpId;
 7     while i < |table| ∧ tileCount < blockSize/nrProcs do
 8         s ← table[i * warpSize + lane];
 9         if isNew(s) then
10             j ← atomicInc(&tileCount);
11             if j < blockSize/nrProcs then
12                 workTile[j] ← s;
13                 markOld(table[i * warpSize + lane]);
14         i ← i + nrOfWarps;
```

As Wijs and Bošnački [42] already discussed, work scanning can be a performance bottleneck, especially when using a large hash table that is only sparsely filled with new states. To combat this problem, GPUexplore already implemented *work claim-*

ing: at the end of each iteration, when copying new states from the cache to global memory, blocks can immediately place these in their work tile. This saves time when scanning for work in the next iteration. In the case that a block manages to fill the entire work tile, it even avoids work scanning completely.

This technique can not be applied during the last iteration of a kernel launch, however, since the contents of shared memory are lost after the termination of a kernel. To solve this problem, we copy the contents of the work tile to global memory after the last iteration. Before the first iteration of the next kernel launch, we copy this information back to the work tile in shared memory.

The second improvement we made is to save the location where the previous scan terminated. During the next iteration, we will start scanning from that location. Suppose the hash table was scanned until location n during the previous iteration. In that case, the locations $0 \leq globalWarpId + k * nrOfWarps < n$, for all $k \in \mathbb{N}$, only contain new states that were inserted between the previous scan and this scan. Therefore, it is more efficient to continue scanning at location n.

The third optimization is to track whether work is available for a block: for every block, there is a flag that indicates whether the part of the hash table scanned by that block contains a new state. The flags are stored in global memory, so they can be accessed by all threads. Initially, all flags are set to false. Whenever a new state is inserted into the hash table, the flag of the block corresponding to the location of the state is set to *true*. When a block scans its part of the hash table and finds no new states, it sets its flag to *false*. Blocks will not try to scan the hash table for new states when their flag is set to *false*. This optimization carries some overhead, due to the increased amount of accesses to global memory. However, blocks no longer scan the hash table unnecessarily. The benefit becomes greater as the hash table becomes larger.

The final, small optimization in this area is to avoid scanning the cache (line 10 of Algorithm 2) when the work tile is empty. In that case, no states have been gathered, no successors have been generated and thus the cache is empty.

3.3 Hash Table

Experiments with larger models showed that GPUexplore would quickly run out of space in its hash table. In most cases, only 50% of the hash table was occupied with states. Further analysis proved that these states where poorly spread over the table. This is all due to a high amount of *hash collisions*: the hash function often produces the same hash value for different state vectors. This leads to all states being concentrated on a relatively small part of the hash table.

Therefore, we have improved the hash function (Algorithm 4). The new implementation first sums all integers of the state vector, while applying some bit shifts in between (line 3). Then, it multiplies with a hash constant a, adds a constant b and computes the modulo with respect to P, where P is a large prime number (line 4). Finally, the position in the hash table is computed by taking the modulo with respect to the number of buckets.

We have empirically determined that this hash function offers a good trade-off between runtime and the amount of hash collisions. It leads to a good spread of states in the hash table.

Algorithm 4: Improved hash function

```
   Input: constant a, constant b, constant P, nr_buckets, state[ ]
   Output: hash
1  hash ← 0;
2  foreach s ∈ state[ ] do
3  |  hash ← (hash + s) << 5;
4  hash ← ((a*hash+b)  mod P)  mod nr_buckets;
```

This hash function can slightly improve the runtime of the state-space exploration. Since states are spread better through the hash table, finding a fully occupied bucket on the first try is less likely. Therefore, less rehashing is required. Furthermore, we changed the implementation to avoid costly 64 bit modulo operations whenever possible.

The second improvement related to the hash table is the removal of all single-threaded accesses to the hash table. In the original implementation, whenever the cache was full, a thread may insert states directly into the hash table in global memory by itself. This single-threaded implementation was sensitive to race-conditions, however, resulting in duplicate entries in the hash table. By exchanging data with warp instructions, it is possible to use the warp version of *findOrPut* instead. This implementation provides coalesced access, so it helps to speed-up exploration in those cases where the cache overflows.

3.4 Branch Divergence and Synchronization

GPUexplore contains a lot of branch divergence. Since the algorithms are complex, this is unavoidable in most places. However, there were several critical points in the code where branch divergence could be avoided.

In the original version of GPUexplore, vector groups and warps were not related in any way. This means that a vector group can be part of more than one warp. To bring the threads of a vector group in sync required a call to the `__syncthreads` function from the CUDA API, which forces the whole block to synchronize. This caused a lot of unnecessary waiting. Therefore, we restructured the thread layout, so that a vector group will never cross a warp boundary. A warp may still contain multiple vector groups, however. This allows us to perform most of the communication between threads via warp instructions and to remove all calls to `__syncthreads` from the main loop of the kernel. This greatly speeds up successor generation, since threads spend less time waiting for each other.

3.5 Generating Synchronizing Transitions

The generation of successors (lines 7 and 8 of Algorithm 2) has also been optimized. In the original design of Wijs and Bošnački [42], finding the action with the smallest label index was done by the group leader, which scanned the buffer of each of the threads in the group. Next, the synchronizing transitions for the action with the smallest label index were generated.

First, we replaced the sequential buffer scanning by butterfly reduction through warp instructions (cf. section 2.4) to find the smallest label index. Since we only have to find the minimum within the vector group, this requires $\lceil \log_2 n \rceil$ warp instructions, where n is the number of processes in the network.

The procedure of generating synchronizing transitions according to sync vectors has also been enhanced. Previously, for every applicable sync vector, a leader would be selected that generated the successors following from that vector. In our improved version, all group members read sync vectors in parallel. They iterate through the array of sync vector until all threads in the vector group have found an applicable sync vector or they have reached the end of the array. Then, the successors following from the sync vectors are computed in parallel.

Chapter 4

Partial-Order Reduction

This chapter explains the details of the implementation of partial-order reduction in GPUexplore. First, it discusses how GPUexplore can be extended with partial-order reduction. Then, a formal proof is provided to show that the proposed algorithms are correct.

Before we explain the extensions required to implement partial-order reduction, it is important to note that the search order of GPUexplore is not strictly DFS or BFS. GPUexplore does not implement a stack or a queue to enforce a work order, but gathers states from global memory. Therefore, the search algorithm should be categorized as *general state expanding algorithm* (GSEA) [17]. We satisfy the action ignoring proviso by implementing the GSEA cycle proviso that was introduced by Bošnački et al. [12].

In literature, multiple practical approaches to partial-order reduction have been proposed. The GPU implementation of *ample sets* [37, 24], *cample sets* [8] and *stubborn sets* [38] is discussed below. Sleep sets have not been considered, since they require a large amount of memory for each state.

In the following sections, we will explain how lines 7 and 8 of Algorithm 2 can be adjusted to implement POR.

4.1 Ample-set Approach

The ample-set approach is based on the idea of *safe* actions [24]: an action is safe if it is independent of all actions of all other processes. While exploring a state s, if there is a process $\Pi[i]$ that has only safe actions enabled in s, then $r(s) = enabled_i(s)$ is a valid ample set, where $enabled_i(s)$ is the set of actions of process $\Pi[i]$ enabled in s. Otherwise, $r(s) = enabled(s)$. In our context of an LTS network, only local

Algorithm 5: Successor generation under the ample-set approach

```
   Data: __global__ table[ ]
   Data: __shared__ cache[ ], buf[ ][ ], reduceProc[ ]
 1 function generateSuccessors():
 2     bufCount ← 0, reduceProc[vgid] ← numProcs;
 3     if processHasOnlyLocalTrans(s, vgtid) then
 4         foreach t ∈ succ_vgtid(s) do
 5             location ← storeInCache(t);
 6             buf[tid][bufCount] ← location;
 7             bufCount ← bufCount + 1;
 8     foreach i ∈ [0..bufCount − 1] do
 9         j ←findGlobal(cache[buf[tid][i]]);
10         if j = NotFound ∨ isNew(table[j]) then
11             atomicMinimum(&reduceProc[vgid], vgtid);
12     __syncthreads();
13     if reduceProc[vgid] < numProcs ∧ reduceProc[vgid] ≠ vgtid then
14         foreach i ∈ [0..bufCount − 1] do
15             markOld(cache[buf[tid][i]]);
16     __syncthreads();
17     if reduceProc[vgid] = vgtid then
18         foreach i ∈ [0..bufCount − 1] do
19             markNew(cache[buf[tid][i]]);
20     if reduceProc[vgid] ≥ numProcs then
21         /* generate the remaining successors                    */
```

actions are safe, so reduction can only be applied if we find a process with only local actions enabled.

An outline of the GPU ample-set algorithm can be found in Algorithm 5. First, the successors of processes that have only local actions enabled are generated. These states are stored in the cache (line 5) by some thread i, and their location in the cache is stored in a buffer that has been allocated in shared memory for each thread (line 6). Then, line 9 finds the location of the states in global memory. This step is performed by threads cooperating in warps to ensure coalesced memory accesses. If the state is not explored yet (line 10), then the cycle proviso has been satisfied (cf. section 2.3.2) and thread i reports it can apply reduction through the *reduceProc* shared variable (line 11). In case the process of some thread has been elected for reduction ($reduceProc[vgid] < numProcs$), the other threads apply the reduction by marking successors in their buffer as old in the cache, so they will not be copied to global memory later. Finally, threads corresponding to elected processes get a

chance to mark their states as new if they have been marked as old by a thread from another vector group (line 19). In case no thread can apply reduction, the algorithm continues as normal (line 21). Why states need to be marked as new will be explained in section 4.4.

4.2 Cample-set Approach

In our definition of a network of LTSs, local actions represent internal process behaviour. Since most practical models frequently perform communication, they have only few local actions and consist mainly of synchronizing actions. The ample-set approach relies on local actions to achieve reduction, so it often fails to reduce the state space. To solve this issue, we implemented *cluster-based* POR [9]. Contrary to the ample-set approach, all actions of a particular set of processes (the *cluster*) are selected. The notion of safe actions is still key. However, the definition is now based on clusters. An action is safe with respect to a cluster $\mathcal{C} \subseteq \{1, \ldots, n\}$ (n is the number of processes in the network), if it is part of a process of that cluster and it is independent of all actions of processes outside the cluster. Now, for any cluster $\mathcal{C}$ that has only actions enabled that are safe with respect to $\mathcal{C}$, $r(s) = \bigcup_{i \in \mathcal{C}} enabled_i(s)$ is a valid cluster-based ample (*cample*) set. Note that the cluster containing all processes always yields a valid cample set.

Whereas Basten and Bošnački [9] determine a tree-shaped cluster hierarchy a priori and by hand, our implementation computes a cluster on-the-fly for every individual state. This should lead to better reductions, since the fixed hierarchy only works for parallel processes that are structured as a tree. Dynamic clustering works for any structure, for example ring or star structured LTS networks. In [9], it is argued that computing the cluster on-the-fly is an expensive operation, so it should be avoided. Our approach, when exploring a state s, is to compute the smallest cluster $\mathcal{C}$, such that $\forall i \in \mathcal{C} : C[i] \subseteq \mathcal{C}$, where $C[i]$ is the set of processes that process i synchronizes with in the state s. This can be done by running a simple fixed-point algorithm, with complexity $O(n)$, once for every $C[i]$ and finding the smallest from those fixed points. This gives a total complexity of $O(n^2)$. However, in our implementation, n parallel threads each compute a fixed point for some $C[i]$. Therefore, we are able to compute the smallest cluster in linear time with respect to the amount of processes. Dynamic clusters do not influence the correctness of the cample-set approach, the reasoning of [9] still applies.

The computation of dynamic clusters is detailed in Algorithm 6. Each vector group accesses two arrays in shared memory: *cluster* to track the dynamic cluster as computed by each thread and *proviso* to track which processes/threads satisfy the

cycle proviso (we remind the reader that each thread is associated with exactly one process). Each thread starts by assigning the information obtained during successor generation to *cluster* and *proviso*(lines 2-3). After the work set of processes is initialized (line 5), some process i is selected from the work set (line 7). Then, i is added to the cluster of this thread (line 8) and the work set and proviso status is updated with the data from i (lines 9-10). When every thread has finished computing the closure, the results are shared (lines 12-13). From all clusters, the group leader selects the smallest (line 16). This cluster is then returned by all threads in the group.

Algorithm 6: Algorithm for computing a cluster in parallel.

```
   Data: __shared__ cluster[ ], proviso[ ]
 1 function determineCluster(myCluster, provisoSatisfied):
 2     cluster[vgtid] ← myCluster;
 3     proviso[vgtid] ← provisoSatisfied;
 4     __syncthreads();
 5     clWork ← myCluster;
 6     while clWork ≠ ∅ do
 7         clWork ← clWork \ {i} for some i ∈ clWork;
 8         myCluster ← myCluster ∪ {i};
 9         clWork ← clWork ∪ (cluster[i] \ myCluster);
10         provisoSatisfied ← provisoSatisfied ∨ proviso[i];
11     __syncthreads();
12     cluster[vgtid] ← myCluster;
13     proviso[vgtid] ← provisoSatisfied;
14     __syncthreads();
15     if vgtid = 0 then
16         selectedCluster ← cluster[i] where proviso[i] ∧ |cluster[i]| ≤ |cluster[j]| for all j;
17         cluster[0] ← selectedCluster;
18     __syncthreads();
19     return cluster[0];
```

The algorithm for generating successors following from the cample-set suffers from the fact that it is not possible to determine a good upper bound on the maximum amount of successors that can follow from a single state. Therefore, it is not possible to statically allocate a buffer, as was done for the ample-set approach (Algorithm 5). Dynamic allocation in shared memory is not supported by CUDA. The only alternative is to check immediately whether the last generated state is marked as *new* in global memory. This is outlined in Algorithm 7. On lines 4 to 6, successors are generated and stored in the cache as usual. Each successor is immediately looked up in the global hash table (line 7). If it has not been explored yet, the cycle proviso is satisfied by the successors of the process associated with the thread (line 9). For

all actions, we determine which processes may synchronize on that action and save this information in *myCluster* (line 10). Then, the cluster is computed according to Algorithm 6 (line 11). Finally, states are marked as *old* (line 14) or *new* (line 14) depending on whether they follow from an action in the cample set.

Algorithm 7: Algorithm for generating states using a cample set.

```
   Data: __shared__ cluster[ ]
 1 function generateSuccessors():
 2     myCluster ← {vgtid};
 3     provisoSatisfied ← false;
 4     foreach a ∈ enabled(s, vgtid) do
 5         foreach t ← succ(s, a) do
 6             loc ← storeInCache(t);
 7             j ←findGlobal(t);
 8             if j = NotFound ∨ isNew(table[j]) then
 9                 provisoSatisfied ← true;
10         myCluster ← myCluster ∪ actionSyncsWith(a);
11     cluster ← determineCluster(myCluster,provisoSatisfied);
12     if vgtid ∈ cluster then
13         foreach t ∈ successors(s, vgtid) do
14             markOld(cache[findInCache(t)]);
15     __syncthreads();
16     if vgtid ∉ cluster then
17         foreach t ∈ successors(s, vgtid) do
18             markNew(cache[findInCache(t)]);
```

4.3 Stubborn-set Approach

The stubborn-set approach was originally introduced by Valmari [38]. The algorithm starts by selecting one enabled action and builds a stubborn set by iteratively adding actions. For enabled actions, all actions that are dependent on that action are added. For disabled actions, all actions that can enable it are added. When a closure has been reached, all enabled actions in the stubborn set together form a persistent set.

Our implementation uses bitvectors to store the stubborn set in shared memory. In case we apply the cycle proviso, we need four of such bitvectors: to store the stubborn set, the work set, the set of enabled actions and the set of actions that satisfy the cycle proviso. This design may have an impact on the practical applicability of the

algorithm, since the amount of shared memory required is relatively high. However, this is the only design that results in an acceptable computational overhead.

To compute a stubborn set, we use several matrices that have been computed during pre-processing. Information about the dependency of actions is stored in the *do-not-accord* (DNA) matrix and information about how action can be enabled and disabled is stored in the *necessary enabling set* (NES) and *necessary disabling set* (NDS) matrices respectively. For more details on these matrices, we refer the reader to Laarman et al. [30]. To further reduce the size of the computed stubborn set, we apply the heuristic function from [30]. Contrary to their implementation, we do not compute a stubborn set for all possible choices of initial action. Effectively, we sacrifice some reduction potential in order to minimize the runtime overhead and the amount of memory required for computing a stubborn set.

An outline of the implementation is given in Algorithm 8. Since it is not possible to determine whether an action is enabled in constant time, we generate the set of enabled transitions before computing the stubborn set (lines 8-9). This also allows us to build a set of actions that satisfy the cycle proviso (lines 10-13). If no action satisfies the cycle proviso, the set of all actions is returned (lines 15-17). On line 20, the group leader selects a single initial action. Our implementation first looks for local actions, because they are less likely to cause large stubborn sets. The actual stubborn set is computed in the loop starting on line 23. Since this is done in parallel, each thread gathers an action from one of its own slots of the work set (line 24). Actions from the DNA matrix or a NES/NDS matrix are added on line 28 and 30 respectively. At the end of a loop, threads vote whether they may still have work to do (line 36). If not, the `buildStubbornSet` function terminates and all states following from actions in the stubborn set are generated (lines 3-6).

Algorithm 8: Algorithm for generating states using a stubborn set.

```
   Data: __shared__ stubborn[ ][ ], work[ ][ ], enabled[ ][ ], proviso[ ][ ], continue[ ]
 1 function generateSuccessors():
 2     buildStubbornSet();
 3     foreach a ∈ enabled(s, vgtid) do
 4         if stubborn[vgid][a] then
 5             foreach t ← succ(s, a) do
 6                 storeInCache(t);
 7 function buildStubbornSet():
 8     foreach a ∈ enabled(s, vgtid) do
 9         enabled[vgid][a] ← true;
10         foreach t ← succ(s, a) do
11             j ←findGlobal(t);
12             if j = NotFound ∨ isNew(table[j]) then
13                 proviso[vgid][a] ← true;
14     __syncthreads();
15     if ¬proviso[vgid][a] for all a then
16         stubborn[vgid][a] ← true for all a;
17         return;
18     __syncthreads();
19     if vgtid = 0 then
20         work[vgid][a] ← true for some a : proviso[vgid][a];
21         continue[vgid] ← true;
22     __syncthreads();
23     while continue[vgid] do
24         act ← a for some a : work[vgid][a] ∧ a = k ∗ num_procs + vgtid;
25         work[vgid][a] ← false;
26         stubborn[vgid][a] ← true;
27         if enabled[vgid][a] then
28             atomicOr(&work[vgid], fetchTexture(DNA[a])∧¬stubborn[vgid]);
29         else
30             atomicOr(&work[vgid], fetchTexture(N[a])∧¬stubborn[vgid]) where
                N ← find_nes_heur(act);
31         __syncthreads();
32         if vgtid = 0 then
33             continue[vgid] ← false;
34         __syncthreads();
35         if act then
36             continue[vgid] ← true;
37         __syncthreads();
```

4.4 Proof of Correctness

In this section, we give a formal argument why the GPU POR algorithms we proposed are correct, i.e. why they produce a reduction function that satisfies the criteria of a persistent set.

4.4.1 Sequential correctness

First, we will prove that each of the POR algorithms we proposed produces valid persistent sets when run with minimal parallelism.

Theorem 1. *Algorithm 5 produces a correct ample set that satisfies the action ignoring proviso when run with minimal parallelism (using only one vector group of* numProcs *threads).*

Proof. The algorithm can generate successors according to either of two results:

- All transitions of a process $\Pi[i]$ that has at least one local transition and no synchronizing transitions ($r(s) = enabled_i(s)$).
- All transitions ($r(s) = enabled(s)$)

Since conditions C0 - C2 are trivially true for the second result, we will focus on the first outcome. C0a and C0b are again trivially true. In order to determine locally whether $r(s) = enabled_i(s)$ satisfies condition C1, we have to satisfy two conditions [2]:

C1a: Any $\alpha \in A[j]$ is independent of $enabled_i(s)$ for $i \neq j$.

C1b: Any $\alpha \in A[i] \setminus enabled(s)$ may not become enabled through the activities of some process $\Pi[j]$ with $i \neq j$.

Because we chose a process with only local actions, all actions in $enabled_i(s)$ are independent of any $\alpha \in A[j]$ (C1a is satisfied). C1b is trivially satisfied: since $enabled_i(s)$ does not contain blocked actions, there is no action in $\Pi[i]$ that can become enabled in any way. Therefore, we have satisfied C1b. We can now deduce that condition C1 is also satisfied.

Condition C2 is satisfied by the fact that *reduceProc* is only changed by threads that have at least one non-closed state in their buffer (line 11). So, the set of successor states resulting from the selected ample set always contains at least one state that is not in closed.

These conditions together imply that the selected set of actions is indeed a correct persistent set (conditions C0 - C2) that satisfies the action ignoring proviso (condition C2). □

As stated in the previous proof, we apply the local criteria from Baier and Katoen [2] for condition C1 to compute a valid ample set. In order to apply these criteria to cample sets, a slight generalization is needed. The following local criteria ensure that condition C1 is satisfied when we choose $r(s) = \bigcup_{i \in C} \mathit{enabled}_i(s)$ for some cluster $\emptyset \neq C \subseteq \{1, \ldots, n\}$ of a network $\mathcal{N}$.

C1c: Any $\alpha \in A[j]$ is independent of $\bigcup_{i \in C} \mathit{enabled}_i(s)$ for $j \notin C$.

C1d: Any $\alpha \in (\bigcup_{i \in C} A[i]) \setminus \mathit{enabled}(s)$ may not become enabled through the activities of some process $\Pi[j]$ with $j \notin C$.

The accompanying lemma is also adapted (Baier and Katoen [2], lemma 8.27):

Lemma 1. *If C1c and C1d hold, then* $r(s) = \bigcup_{i \in C} \mathit{enabled}_i(s)$ *for some cluster* $\emptyset \neq C \subseteq \{1, \ldots, n\}$ *satisfies condition C1 for all executions that start in* s*.*

Proof. In case $C = \{1, \ldots, n\}$, C1 trivially holds. In other cases, the reasoning below applies.

Suppose that C1c and C1d hold, but C1 does not hold. Let s be the state we are exploring. Because C1 does not hold, there exists an execution $s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_m} s_m \xrightarrow{\alpha_{m+1}}$, where $\alpha_1, \ldots, \alpha_m \notin r(s)$ (and therefore $\alpha_1, \ldots, \alpha_m \notin \bigcup_{i \in C} \mathit{enabled}_i(s)$) and α_{m+1} depends on $r(s)$. Because of condition C1c, any action that depends on $r(s)$ is an action of some process $\Pi[i]$ with $i \in C$.

Let k be the largest index in $\{1, \ldots, m\}$ such that $\alpha_1, \ldots, \alpha_{k-1}$ are actions of processes not in C: $\alpha_1, \ldots, \alpha_{k-1} \in \bigcup_{j \notin C} A[j] \setminus \bigcup_{i \in C} A[i]$ and $\alpha_k \in \bigcup_{i \in C} A[i]$. Because the actions $\alpha_1, \ldots, \alpha_{k-1}$ cannot change the state of any process $\Pi[i]$ with $i \in C$, states $s_1, \ldots, s_{k-1}$ are the same as state s in all slots $i \in C$. $\alpha_k \notin r(s)$ and $\alpha_k \in \bigcup_{i \in C} \mathit{enabled}_i(s_{k-1})$, so action α_k becomes enabled in some slot $i \in C$ through the execution of one of the actions $\alpha_1, \ldots, \alpha_{k-1}$. Since $\alpha_1, \ldots, \alpha_{k-1}$ are actions of processes $\Pi[j]$ with $j \notin C$, this contradicts C1d. □

Theorem 2. *Algorithm 7 produces a correct cample set that satisfies the action ignoring proviso when run with minimal parallelism (using only one vector group of* numProcs *threads).*

Proof. For this proof, we can follow exactly the same reasoning as for Theorem 1, but we use local criteria C1c and C1d instead of C1a and C1b. □

Theorem 3. *Algorithm 8 produces a correct stubborn set that satisfies the action ignoring proviso when run with minimal parallelism (using only one vector group of* numProcs *threads).*

Since our stubborn-set algorithm is not fundamentally different from the original sequential version, we refer to Theorem 4.18 of Godefroid [21] for the proof of this theorem.

4.4.2 Parallel correctness

The cache that is maintained in shared memory is read and modified by all threads. Therefore, our algorithms may no longer be correct when executed on multiple vector groups within one block. With the following theorem, we prove that this is not the case. The ample-set algorithm is used as an example, but the reasoning applies to all three implementations.

Theorem 4. *Algorithm 5 produces a correct ample set when run using multiple vector groups within one block.*

Proof. We consider two vector groups A and B that are member of the same block. From Theorem 1 we know that the result of each of the vector groups produces a correct ample set when run sequentially. When run in parallel, there are two ways in which the actions of group B can influence the result of group A.

- Vector group A does not apply reduction. However, group B marks some of the states in the buffer of group A as *old*. This may lead to an unwanted loss of states.
- Vector group A applies reduction by marking states in the cache as *old* (line 15). However, group B later marks some or all of these states as *new* (line 19). This makes the group of selected transitions for A larger, possibly affecting correctness.

In the first situation, the actions of group B are negated, because group A marks all the successors it found as *new* after B marked them as *old*. Therefore, the correctness of the result of group A is not affected.

The second situation is the reverse of the first one. Here, B adds successors to the result computed by A. This leads to the same states being added to *Closed* as when group A and B would be run in sequence. Therefore, the correctness of the algorithm is not affected. □

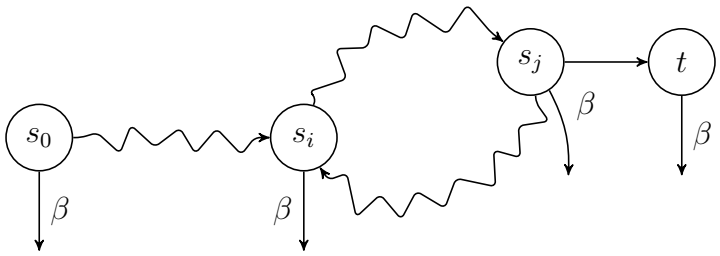


Figure 4.1: 'Lasso' shaped path from the proof of Lemma 2

Blocks communicate with each other via global memory. For partial-order reduction, the only relevant access to global memory happens at line 10 of Algorithm 5. We will now show that the correctness of our algorithms is not affected when running on multiple blocks. First, we introduce a new version of the cycle proviso and show that it implies the action-ignoring proviso.

Lemma 2. (closed-set cycle proviso) *If a reduction algorithm satisfies conditions C0a, C0b and C1 and for every cycle* $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} s_n \xrightarrow{\alpha_n} s_0$ *in the reduced state space with* $\beta \in \mathit{enabled}(s_0)$ *and* $\beta \neq \alpha_i$ *for all* $0 \leq i \leq n$*, selects* (i) *at least one transition labelled with* β *or* (ii) *at least one transition that, during the generation of the reduced state space, led to a state outside the cycle that has not been explored yet (i.e.* $\exists\, i\, \exists (s_i, \gamma, t) \in \tau : \gamma \in r(s_i) \wedge t \notin$ Closed*); then condition C2ai is satisfied.*

Proof. Suppose that action $\beta \in \mathit{enabled}(s_0)$ for some $s_0 \in S_r$ is always ignored, i.e. condition C2ai is not satisfied. This means there is no execution $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} s_n \xrightarrow{\beta} t$ where $\alpha_i \in r(s_i)$ for all $0 \leq i < n$. Because we are dealing with finite state spaces, every execution that infinitely ignores β has to end in a cycle. These executions have a 'lasso' shape, they consist of an initial phase and a cycle. Let $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{i-1}} s_i \xrightarrow{\alpha_i} \ldots \xrightarrow{\alpha_{n-1}} s_n \xrightarrow{\alpha_n} s_i$ be the execution with the longest initial phase, i.e. with the highest value i (see Figure 4.1). Since condition C1 is satisfied, β is independent of any α_k and thus enabled on any s_k with $0 \leq k \leq n$. It is assumed that for at least one of the states $s_i \ldots s_n$ an action exiting the cycle is selected. Let s_j be such a state ($i \leq j \leq n$). Since β is ignored, $\beta \notin r(s_j)$. According to the assumption, one of the successors found through $r(s_j)$ has not been in *Closed*. Let this state be t. Any finite path starting with $s_0 \ldots s_j t$ cannot end in a deadlock without taking action β at some point (condition C0b). Any infinite path starting with $s_0 \ldots s_j t$ has a longer initial phase (after all $j + 1 > i$) than the execution we assumed had the longest initial phase. Thus, our assumption is contradicted. □

Whereas the closed-set proviso (cf. section 2.3.2) is a local condition which can be

evaluated on a per-state basis, the new *closed-set cycle* proviso is a global property. Although the new proviso assumes C0a, C0b and C1, it can allow smaller reduction functions r, since only one transition per cycle is required to lead to a state outside *Closed*.

Before we start the next proof, it is important to note three things. Firstly, the work gathering function on line 4 of Algorithm 2 moves the gathered states from *Open* to *Closed*. Secondly, the working of the algorithm with respect to conditions C0a, C0b and C1 is not affected when it executes on multiple blocks. Finally, we again use our ample-set algorithm as an example, but the theorem applies to all three algorithms.

Theorem 5. *Algorithm 5 produces a correct ample set that satisfies the cycle proviso when run on multiple blocks.*

Proof. Let $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-2}} s_{n-1} \xrightarrow{\alpha_{n-1}} s_0$ be a cycle in the reduced state space. In case α_0 is dependent on all other enabled actions in s_0, there is no action to be ignored and C2ai is satisfied.

In case there is an action in s_0 that is independent of α_0, this action is prone to ignoring. Let us call this action β. Because condition C1 is satisfied, β is also enabled in the other states of the cycle: $\beta \in \mathit{enabled}(s_i)$ for all $0 \leq i < n$.

We now consider the order in which states on the cycle can be explored by multiple blocks. Let s_i be one of the states of this cycle that is gathered from *Open* first (line 4, Algorithm 2). There are two possibilities regarding the processing of state s_{i-1}:

- It is gathered from *Open* at exactly the same time as s_i. When the processing for s_{i-1} arrives at line 10 of Algorithm 5, it will find s_i in *Closed*.
- It is gathered later than s_i. Again, s_i will be in *Closed*.

Since s_i is in *Closed* in both cases, at least one other action will be selected for $r(s_{i-1})$. If all successors of s_{i-1} are in *Closed*, then β has to be selected. Otherwise, at least one transition to a state that is not in *Closed* will be selected. Now we can apply Lemma 2. □

Combining the obtained results gives us the following corollary:

Corollary 1. *Algorithms 5, 7 and 8 produce a correct persistent set that satisfies the cycle proviso when run with multiple vector groups on multiple blocks.*

Proof. By combining the results of Theorem 4 and Theorem 5, we can deduce that the algorithms produce a correct persistent set when run with any number of vector groups distributed over any number of blocks. The generated persistent set satisfies the action-ignoring proviso. □

Chapter 5

Experiments

This chapter presents the results of several experiments. First, it shows the speed-up gained by the optimizations proposed in Chapter 3. Second, the performance of the proposed POR algorithms is determined.

5.1 Speed-up of Optimizations

The optimizations that we proposed in Chapter 3 have been implemented in GPUexplore. We call this version GPUexplore 2.0[1]. We want to determine the speed-up offered by the optimizations over the original version from [42]. Additionally, we will compare the performance with a traditional sequential model checker.

The models that were used as benchmarks (22 in total) have different origins. `Cache`, `sieve`, `odp`, `transit`, `asyn3` and `des` are all EXP models from the examples included in the CADP toolkit [20]. The `leader_election`, `anderson`, `lamport`, `lann`, `peterson` and `szymanski` models come from the BEEM database[2] and have been translated from DVE to EXP. `1394`, `acs` and `wafer stepper` are originally mCRL2 [15] models and have also been translated by hand to EXP. `broadcast` has been created by Wijs and Bošnački [42]. The models with a .1-suffix are enlarged versions of the original models [42]. The details of the models can be found in Table 5.1. 'stub. set size' indicates the maximum size of the stubborn set, which is relevant for the POR experiments. Since the stubborn set size mainly depends on the synchronization rules, it also gives an indication of the amount of synchronization rules in the network.

[1] Sources are available from https://github.com/ThomasNeele/GPUexplore

[2] http://paradise.fi.muni.cz/beem/

Table 5.1: Overview of the models used in the benchmarks

model	#states	#transitions	stub. set size
cache	616	4,631	222
leader_election1	4,261	12,653	4,712
acs	4,764	14,760	134
sieve_10	23,627	84,707	941
odp	91,394	641,226	464
1394	198,692	355,338	301
acs.1	200,317	895,004	139
transit	3,763,192	39,925,524	73
wafer_stepper.1	3,772,753	19,028,708	880
odp.1	7,699,456	31,091,554	556
1394.1	10,138,812	96,553,318	300
asyn3	15,688,570	86,458,183	1,315
anderson6	18,206,917	86,996,322	786
lamport8	62,669,317	304,202,665	305
des	64,498,297	518,438,860	12
szymanski5	79,518,740	922,428,824	481
broadcast	105,413,504	1,264,962,048	70
peterson7	142,471,098	626,952,200	2,880
lann6	144,151,629	648,779,852	48
lann7	160,025,986	944,322,648	48
asyn3.1	190,208,728	876,008,628	1,363

For all our experiments with GPUexplore 2.0, we use an NVIDIA Titan X, which was released in 2015. The Titan X has 24 SMs each with 128 CUDA cores, giving a total of 3072 CUDA cores. Each SM has 96KB of shared memory and the global memory has a size of 12GB. We allocate 5GB of the global memory for the hash table.

Before we compare our implementation to other implementations, we have to determine which parameter values give the best performance. There are two parameters that can be tuned for optimal performance: the amount of iterations per kernel launch (K) and the amount of blocks (B). We fix the amount of threads per block (T) to 512, since any other value reduces the occupancy of the GPU.

We start by determining the optimal number of iterations per kernel launch. We run GPUexplore with 6144 blocks and 5GB allocated for the hash table and vary K between 1 and 32. The results can be found in Figure 5.1. When varying K between 1 and 4, the runtime for the majority of models does not change significantly. For higher K up to 32, a clear upward trend in runtime is visible. Only `odp.1` and `des` gain a significant speed-up with K around 8. `des` is structurally different from the

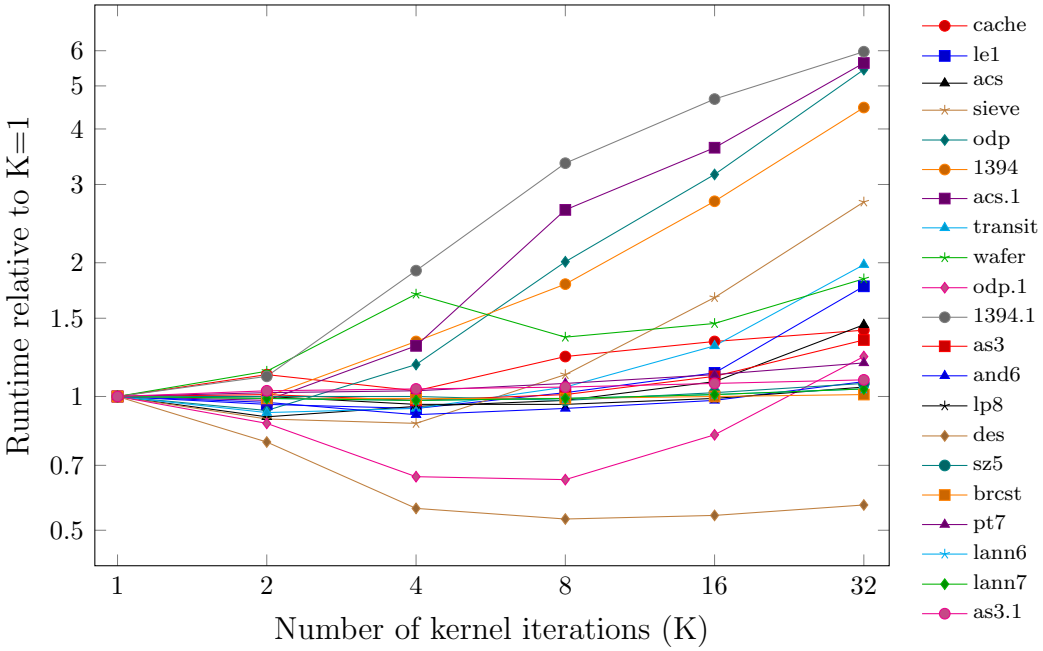


Figure 5.1: Runtime while varying the amount of kernel iterations (6144 blocks)

other models: it contains one very large process and several smaller ones. For the following experiments, we will fix K to one, since it gives the best performance for most of the models.

The fact that a high value for K causes a high runtime can be explained by the synchronizing effect of a kernel launch: at the end of a launch, the CPU waits until all GPU blocks are finished executing. At the beginning of the next launch, all memory writes from previous launches are guaranteed to be visible. This reduces the amount of work scanning needed. There is no function available in the CUDA API to replace the synchronization offered by a kernel launch. In [42], ten iterations per kernel launch yielded optimal performance. Since we have reduced the overhead of a kernel launch by temporarily storing the work tile in global memory (cf. section 3.2), the performance can be increased by performing a kernel launch more often, and thus synchronize the blocks more often.

Next, we will optimize the amount of blocks that the kernel runs on. With our kernel, an SM requires two blocks to be fully occupied and since the Titan X has 24 SMs, we vary B from 48 to 49152. The results are plotted in Figure 5.2.

It is clear from the data that 48 blocks are not enough to fully use the processing power of the GPU. More blocks are needed to hide the memory latency. An additional

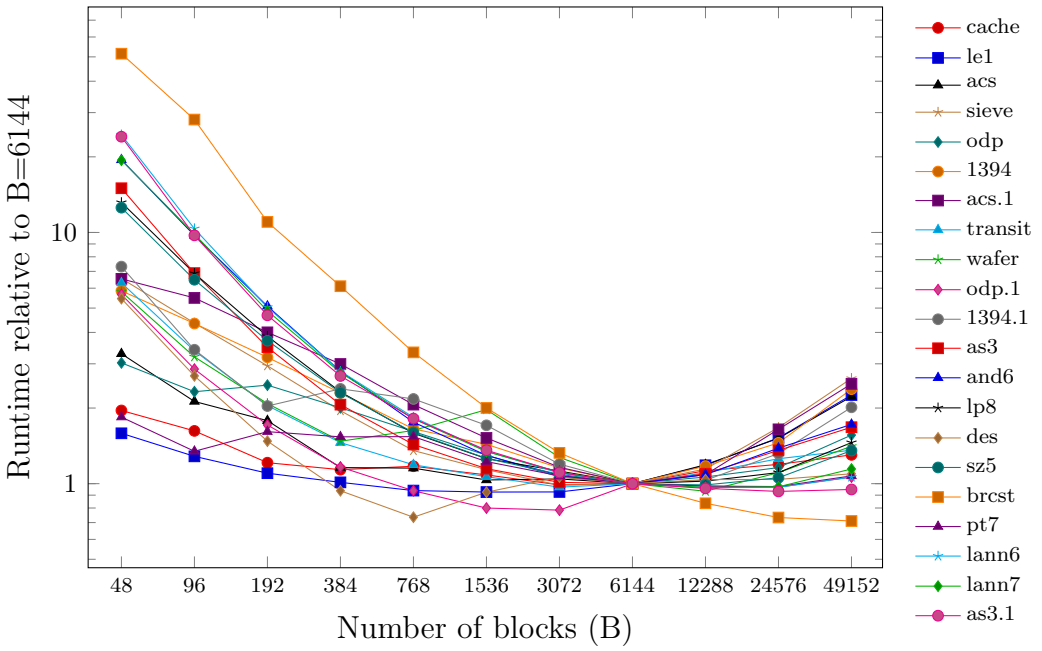


Figure 5.2: Runtime while varying the amount of blocks (1 kernel iteration)

factor influencing the speed-up, is the fact that the availability of work is tracked on a per-block basis. As we increase the amount of blocks, each work available flag covers a smaller part of the hash table. Fine-grained work available flags help to prevent more unnecessary work scanning. When the amount of blocks is very high, the distribution of work between blocks becomes uneven. This especially affects the small models. Only the `broadcast` model benefits from a high amount of blocks. From these results, we can conclude that 6144 blocks and one iteration per kernel launch are the parameters that give the best performance.

Since the `odp.1` and `des` model show different behaviour from the other models when varying either parameter, we will further study the parameter space for those models. The results are displayed in Figure 5.3. The best result for `odp.1` (1.36 seconds) is achieved with 6144 blocks and eight kernel iterations. This is 37% faster than the result with our standard parameters (B=6144, K=1). For the `des` model, GPUexplore performs best when running on 12288 blocks and eight kernel iterations per launch. The runtime is 11.82 seconds, which is 51% less than the runtime with standard parameters.

We compare our implementation directly with the implementation presented in [42]. We also compare with the combination of the EXP.OPEN and Generator

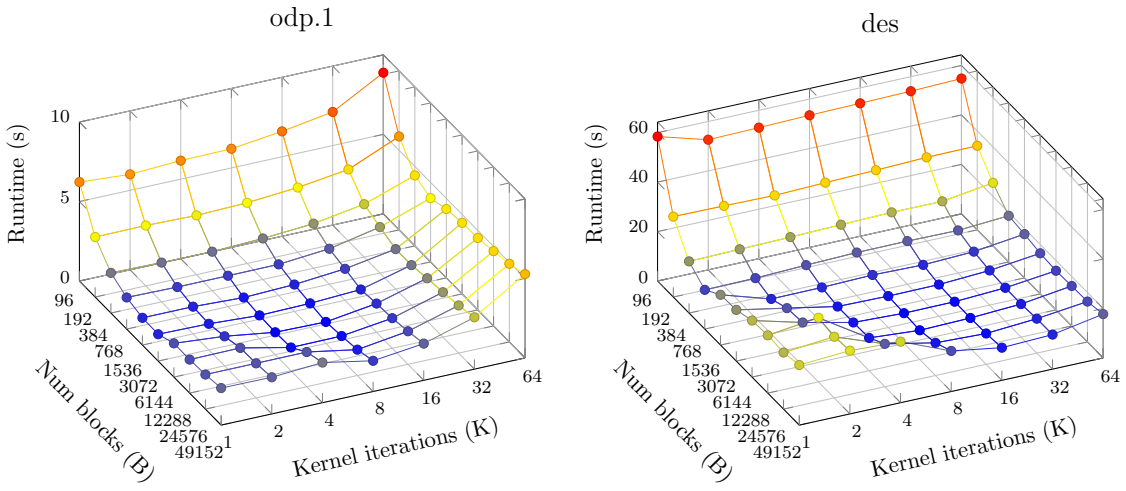


Figure 5.3: Runtime of `odp.1` and `des` for all combinations of K and B

tools from CADP [20]. These tools run on a single thread. Since the EXP language originates from CADP, it is the preferred tool to compare with. We also considered comparing our implementation to the tool from [44]. However, their tool does not use a standardized input language. We were unable to convert our models to their input format.

CADP is run on an Intel Core i5 3350P with 8GB of RAM. For reference, we also include the original results of GPUexplore from [42], that were obtained with a NVIDIA K20m (released in 2012). This GPU has 13 SMs and 5GB of global memory. Since it is not possible to measure the runtime of only the state-space exploration for CADP, we measure the total runtime of each program (initialization and exploration). For the original version of GPUexplore, we run the kernel on 3120 blocks of 512 threads and perform ten iterations per kernel launch, since this gives the best performance [42]. For GPUexplore 2.0, we assign 6144 blocks of 512 threads and perform only one iteration per kernel launch. All GPU experiments are repeated five times. The CPU experiments are run a single time. The results are presented in Table 5.2. Runtimes are reported in seconds. The two columns under 'speed-up' indicate the speed-up of GPUexplore 2.0 over CADP and GPUexplore on the Titan X respectively.

GPUexplore 2.0 manages to bring all runtimes down to under 37 seconds, whereas CADP takes around an hour for several models. On average, GPUexplore 2.0 is 70 times faster than CADP and 7.8 times faster than the original GPUexplore. The new hardware brings a considerable speed-up over the K20m: on average the Titan X is

Table 5.2: Runtime in seconds for CADP, the original GPUexplore and GPUexplore 2.0.

	CADP	GPUexplore-original		GPUexplore 2.0	speed-up	
model	CPU	K20m	Titan X	Titan X	seq.	orig
acs	2.25	10.51	2.26	0.33	6.9	6.9
odp	2.03	8.63	2.19	0.34	5.9	6.4
1394	2.10	23.10	3.85	0.51	4.1	7.6
acs.1	3.58	15.06	2.77	0.46	7.8	6.0
transit	37.79	26.20	4.54	1.21	31.3	3.8
wafer_stepper.1	22.25	47.25	7.33	1.42	15.7	5.2
odp.1	76.73	29.78	5.78	1.84	41.8	3.1
1394.1	66.33	61.40	8.44	1.90	34.8	4.4
asyn3	352.56	273.41	37.97	3.87	91.2	9.8
lamport8	944.80	221.80	41.30	6.91	136.7	6.0
des	468.51	107.22	25.42	18.64	25.1	1.4
szymanski5	1393.35	512.13	86.17	8.93	156.0	9.6
peterson7	3463.06	4337.41	1004.07	36.42	95.1	27.6
lann6	2377.73	492.70	94.85	12.52	189.9	7.6
lann7	3035.55	877.74	164.90	19.83	153.1	8.3
asyn3.1	4360.00	2703.61	421.87	36.61	119.1	11.5
average					69.7	7.8

5.4 times faster.

For the smaller models, the speed-up that can be gained by the parallel power of thousands of threads is limited. If the number of state in a *frontier* (BFS-like search layer) is smaller than the number of states that can be processed in parallel, then not all threads are occupied and the efficiency drops. Therefore, GPUexplore achieves a relatively small speed-up over CADP of less than ten for `acs`, `odp`, `1394` and `acs.1`. For larger models, GPUexplore can achieve a speed-up of two orders of magnitude or more.

The least amount of speed-up, relative to the original GPUexplore, is gained for the `des` model. That has two reasons: (i) the parameters we selected are not optimal for `des`, as shown above, and (ii) `des` consists of many local transitions, while most of our optimizations are aimed at speeding-up the computation of synchronizing transitions. In contrast, `peterson7` contains many synchronization rules. Therefore, GPUexplore 2.0 achieves the largest speed-up over the original GPUexplore for this model. It should be noted that, for `peterson7`, the original GPUexplore on a K20m is even slower than CADP.

Despite the improvements, CADP scales better with the amount of synchroniza-

Table 5.3: Time (in seconds) required for state-space exploration (not including initialization phase).

model	GPUexplore-original	GPUexplore 2.0	speed-up
acs	2.04	0.06	33.6
odp	1.97	0.09	21.2
1394	3.61	0.25	14.5
acs.1	2.54	0.21	12.2
transit	4.30	0.94	4.6
wafer_stepper.1	7.10	1.16	6.1
odp.1	5.54	1.58	3.5
1394.1	8.16	1.60	5.1
asyn3	37.73	3.61	10.4

tion rules. `peterson7` and `asyn3.1`, both large models with many synchronization rules, show a smaller speed-up than either `lann` model, which have a very small amount of synchronization rules.

The speed-up measurements are skewed for the smaller models, since, when the state space is small, most of the runtime for GPUexplore 2.0 is spent on initialization. Therefore, we have repeated the experiments, but now we only measure the time required for state-space exploration on the Titan X. Any time required for initialization is excluded. Table 5.3 shows the runtime for the models where the initialization significantly influences the measured speed-up. The average speed-up over all models is now 11.5 times. These results show that our improvements in the area of work scanning pay off most when the model is small and the hash table is sparsely filled with states.

5.2 POR Experiments

We want to determine the potential of applying POR in GPU model checking and how it compares to POR on a multi-core platform. Additionally, we want to determine which POR approach is best suited to GPUs. We will focus on measuring the reduction and overhead of each implementation.

We implemented the proposed algorithms in GPUexplore 2.0. Since GPUexplore only accepts EXP models as input, we added an EXP language front-end to LTSmin [27] to make a comparison with a state-of-the-art multi-core model checker possible. We remark that it is out of the scope of this paper to make an absolute speed comparison between a CPU and a GPU, since it is hard to compare completely different hardware

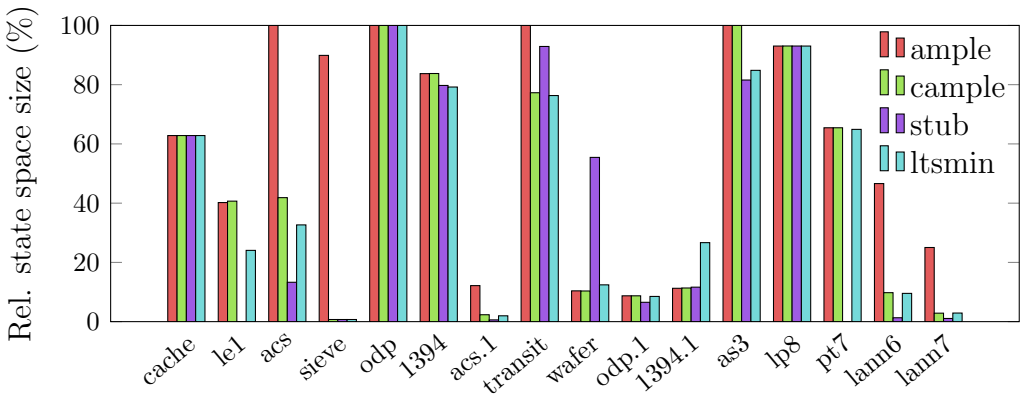


Figure 5.4: State space of POR implementations relative to the full state space.

and tools. Moreover, speed comparisons have already been done before [7, 42, 44].

For these experiments, we used a representative subset of the models from the previous section. For the GPU experiments, we again used the Titan X. GPUexplore was executed on 6144 blocks of 512 threads and performed one iteration per kernel launch, which gives the best performance (cf. section 5.1). LTSmin was benchmarked on a machine with 24GB of memory and two Intel Xeon E5520 processors, giving a total of 16 threads. We used BFS as search order. The stubborn sets were generated by the closure algorithm with a heuristic function to find the best NES [30].

For the first experiment, we disabled the cycle proviso, which is not needed when checking for deadlocks. For each model and for each POR approach, we executed the exploration algorithm 10 times. The average size of the reduced state space relative to the full state space is plotted in the first chart of Figure 5.4 (the full state space has a size of 100% for each model). The error margins are not depicted because they are very small.

The first thing to note is that the state spaces of the `leader_election1` and `peterson7` models cannot be computed under the stubborn-set approach. The reason is that the amount of synchronization rules is very high, so the amount of shared memory required to compute a stubborn set exceeds the amount of shared memory available.

On average, the stubborn-set approach offers the best reduction, followed by the cample-set approach. Only for the `wafer_stepper.1` model, the stubborn-set approach offers a significantly worse reduction. As expected, the cample-set approach always offers roughly similar or better reduction than the ample-set approach, since it is a generalization of the ample-set approach. Overall, the reduction achieved by

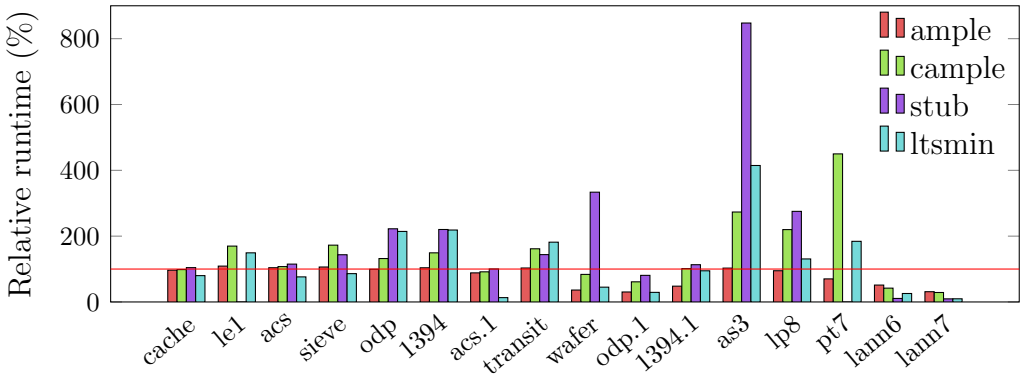


Figure 5.5: Runtime of POR implementations relative to the runtime of full state-space exploration.

GPUexplore and LTSmin is comparable. Note that for GPUexplore, any reduction directly translates into memory saving. For LTSmin, this may not be the case, since its database applies *tree compression* [31].

Additionally, we measured the time it took to generate the full and the reduced state space. To get a good overview of the overhead resulting from POR, the relative performance is plotted in Figure 5.5. For each platform, the runtime of full state-space exploration is set to 100% and is indicated by a red line. Again, the error margins are small, so we do not depict them.

These results show that the ample-set approach induces no significant overhead. For models where good reduction is achieved, it can speed-up the exploration process by up to 3.6 times for the `acs.1` model. On the other hand, the cample and stubborn-set approach suffer from significant overhead. When no or little reduction is possible, this slows down the exploration process by 2.7 times and 8.5 times respectively for the `asyn3` model. This model has the largest amount of synchronization rules after the `leader_election1` and `peterson7` models.

As explained in the previous section, the GPU is not fully occupied when processing small models. Since this problem is only worse under POR, little to no speed-up can be gained when applying POR to small models. When looking at the largest models, the overhead for LTSmin is more that two times lower than for GPUexplore's stubborn-set approach. This shows that our implementation not only has overhead from generating all successors twice, but also from the stubborn-set computation.

In the second experiment, we used POR with cycle proviso. Figure 5.6 shows the relative size of the state space and the relative runtime required. As expected, less reduction is achieved. The checking of the cycle proviso induces only a little

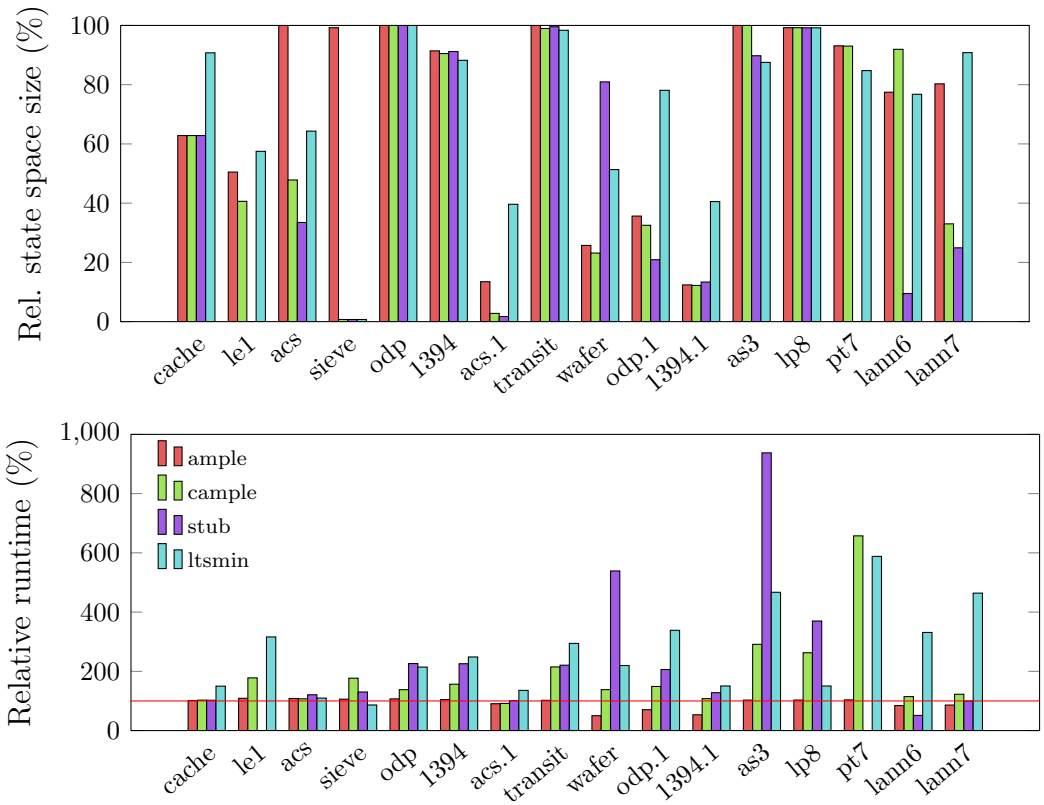


Figure 5.6: State space and runtime of POR implementations with cycle proviso relative to the full state-space exploration.

extra overhead for the ample-set and cample-set approach (not more that 8% and 12% respectively). The extra runtime overhead for the stubborn-set approach can be significant, however: up to 43% for the `transit` model (comparing the amount of states visited per second). When applying the cycle proviso, the reduction achieved by LTSmin is significantly worse. This is due to the fact that LTSmin checks the cycle proviso after generating the smallest stubborn set. If that set does not satisfy the proviso, then the set of all actions is returned. The approach of GPUexplore, where the initial action satisfies the cycle proviso, often finds a smaller stubborn set than LTSmin. Therefore, GPUexplore achieves a higher amount of reduction when applying the cycle proviso.

Table 5.4: Average size of the reduced state space (%)

average size $\mathcal{T}_r$ (%)	ample	cample	stubborn	ltsmin
no proviso	60.26	43.21	42.91	42.26
cycle proviso	71.26	56.84	52.01	71.88

Table 5.4 shows the average size of the reduced state space for each implementation. Since GPUexplore's stubborn-set implementation cannot compute $\mathcal{T}_r$ for `leader_election1` and `peterson7`, those models have been excluded. The performance of the cample-set and stubborn-set implementations is very close to LTSmin.

Chapter 6

Conclusion

In this thesis, we showed that partial-order reduction on a many-core platform can achieve a reduction similar to or better than POR on a multi-core platform. When the cycle proviso is applied, the reduction of our approach is better than the reduction of LTSmin. Our three POR implementations suffer an acceptable amount of runtime overhead, which is mainly caused by the limitations of shared memory.

We proposed an improvement for cluster-based POR, namely dynamic clusters. The benchmarks show that computing clusters on-the-fly imposes barely any runtime overhead, while yielding a reduction similar to the stubborn-set approach. With this improvement, the cample-set approach best suits our goal of reducing memory usage with minimal runtime overhead. Whereas the stubborn-set approach can cause a high amount of runtime overhead, the overhead for the cample-set approach is limited to roughly 100%. In addition, it can be applied to all models and does not require additional preprocessing of the model.

In addition to saving memory with POR, we also proposed several optimizations to save runtime, which together give an average speed-up of 7.8 times over the original version of GPUexplore. Combined with the latest hardware improvements of GPUs, this gives us a total speed-up of 70 times over sequential CADP. For the largest models, the speed-up can exceed two orders of magnitude. Thus, we can bring sequential runtimes of more than one hour down to less than 37 seconds.

Further research into the memory limitations of GPU model checking is necessary. One approach to this problem may be employing multiple GPUs to run the exploration algorithm. This will raise new challenges regarding the implementation of the hash table. It also raises the question how scalable such a multi-GPU implementation is. Another possibility for further study is improving the POR implementation to preserve liveness properties, since checking such properties is supported by the lat-

est version of GPUexplore [39]. Preserving liveness properties under POR requires a stronger version of the cycle proviso and an additional *visibility* condition. We expect that implementing these conditions is straightforward.

Bibliography

[1] CUDA web page. http://www.nvidia.com/object/cuda_home_new.html.

[2] C. Baier and J.-P. Katoen. *Principles of model checking*. MIT Press, 2008.

[3] J. Barnat, P. Bauch, L. Brim, and M. Češka. Designing fast LTL model checking algorithms for many-core GPUs. *Journal of Parallel and Distributed Computing*, 72(9):1083–1097, sep 2012.

[4] J. Barnat, L. Brim, M. Češka, and T. Lamr. CUDA Accelerated LTL Model Checking. In *ICPADS 2009, Proceedings*, number 201, pages 34–41, Shenzhen, China, 2009.

[5] J. Barnat, L. Brim, and P. Ročkai. DiVinE multi-core - A parallel LTL model-checker. In *ATVA 2008, Proceedings*, volume 5311, pages 234–239, 2008.

[6] J. Barnat, L. Brim, and P. Ročkai. Parallel partial order reduction with topological sort proviso. In *SEFM 2010, Proceedings*, number October, pages 222–231, 2010.

[7] E. Bartocci, R. Defrancisco, and S. A. Smolka. Towards a GPGPU-Parallel SPIN Model Checker. In *SPIN 2014, Proceedings*, pages 87–96, San Jose, CA, USA, 2014. ACM.

[8] T. Basten and D. Bošnački. Enhancing partial-order reduction via process clustering. In *ASE 2001, Proceedings*, pages 245–253, 2001.

[9] T. Basten, D. Bošnački, and M. Geilen. Cluster-Based Partial-Order Reduction. *Automated Software Engineering*, 11(4):365–402, 2004.

[10] D. Bošnački, S. Edelkamp, D. Sulewski, and A. Wijs. Parallel probabilistic model checking on general purpose graphics processors. *STTT*, 13(1):21–35, 2010.

[11] D. Bošnački and G. J. Holzmann. Improving Spin's Partial-Order Reduction for Breadth-First Search. In *SPIN 2005, Proceedings*, pages 91–105, San Francisco, CA, USA, 2005.

[12] D. Bošnački, S. Leue, and A. Lluch-Lafuente. Partial-order reduction for general state exploring algorithms. *STTT*, 11(1):39–51, 2009.

[13] M. Češka, P. Pilař, N. Paoletti, L. Brim, and M. Kwiatkowska. PRISM-PSY: Precise GPU-Accelerated Parameter Synthesis for Stochastic Systems. In *TACAS 2016, Proceedings*, volume 9636 of *LNCS*, pages 367–384, Eindhoven, The Netherlands, 2016.

[14] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2001.

[15] S. Cranen, J. F. Groote, J. J. A. Keiren, F. P. M. Stappers, E. P. De Vink, W. Wesselink, and T. A. C. Willemse. An Overview of the mCRL2 Toolset and Its Recent Advances. In *TACAS 2013, Proceedings*, volume 7795 of *LNCS*, pages 199–213, Rome, Italy, 2013.

[16] A. E. Dalsgaard, A. Laarman, K. G. Larsen, M. C. Olesen, and J. Van De Pol. Multi-core reachability for timed automata. In *FORMATS 2012, Proceedings*, volume 7595 of *LNCS*, pages 91–106, 2012.

[17] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Partial-order reduction and trail improvement in directed model checking. *STTT*, 6(4):277–301, 2004.

[18] S. Edelkamp and D. Sulewski. Efficient explicit-state model checking on general purpose graphics processors. In *SPIN 2010, Proceedings*, volume 6349 of *LNCS*, pages 106–123, Enschede, The Netherlands, 2010.

[19] S. Edelkamp and D. Sulewski. External memory breadth-first search with delayed duplicate detection on the GPU. In *Model Checking and Artificial Intelligence*, pages 12–31, Atlanta, Georgia, jul 2010. Springer Verlag.

[20] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *STTT*, 15(2):89–107, 2013.

[21] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. PhD thesis, 1996.

[22] P. Godefroid and P. Wolper. A Partial Approach to Model Checking. *Information and Computation*, 110(2):305–326, 1994.

[23] G. J. Holzmann and D. Bošnački. The design of a multicore extension of the SPIN model checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, 2007.

[24] G. J. Holzmann and D. Peled. An Improvement in Formal Verification. In *FORTE '94, Proceedings*, pages 192–211, 1994.

[25] R. Iosif. Symmetry Reduction Criteria for Software Model Checking. In *SPIN 2002, Proceedings*, volume 2318, pages 22–41, 2002.

[26] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1-2):41–75, 1996.

[27] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk. LTSmin: High Performance Language-Independent Model Checking. In *TACAS 2015, Proceedings*, volume 9035 of *LNCS*, pages 692–707, London, UK, 2015.

[28] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün. Static Partial Order Reduction. In *TACAS 1998, Proceedings*, pages 345–357, Lisbon, Portugal, 1998.

[29] A. Laarman, R. Langerak, J. van de Pol, M. Weber, and A. Wijs. Multi-core nested depth-first search. In *ATVA 2011, Proceedings*, volume 6996 of *LNCS*, pages 321–335. Springer, 2011.

[30] A. Laarman, E. Pater, J. van de Pol, and M. Weber. Guard-Based Partial-Order Reduction. In *SPIN 2013, Proceedings*, volume 7976 of *LNCS*, pages 227–245, Stony Brook, NY, USA, 2013.

[31] A. Laarman, J. van de Pol, and M. Weber. Multi-core LTSmin: Marrying modularity and scalability. In *NASA Formal Methods, Proceedings*, volume 6617 of *LNCS*, pages 506–511, Pasadena, CA, USA, 2011.

[32] A. Laarman and A. Wijs. Partial-Order Reduction for Multi-core LTL Model Checking. In *HVC 2014, Proceedings*, volume 8855 of *LNCS*, pages 267–283, Haifa, Israel, 2014.

[33] F. Lang. Refined interfaces for compositional verification. In *FORTE 2006, Proceedings*, volume 4229 of *LNCS*, pages 159–174. Springer, 2006.

[34] A. Lluch-Lafuente, S. Leue, and S. Edelkamp. Partial Order Reduction in Directed Model Checking. In *SPIN 2002, Proceedings*, volume 2318 of *LNCS*, pages 112–127, Grenoble, France, 2002.

[35] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[36] T. Neele, A. Wijs, D. Bošnački, and J. Van de Pol. Partial-Order Reduction for GPU Model Checking. In *ATVA 2016, proceedings*, LNCS, accepted for publication, 2016.

[37] D. Peled. All from one, one for all: on model checking using representatives. In *CAV 1993, Proceedings*, volume 697, pages 409–423, Elounda, Greece, 1993.

[38] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets*, volume 483, pages 491–515, 1991.

[39] A. Wijs. BFS-Based Model Checking of Linear-Time Properties With An Application on GPUs. In *CAV 2016, Proceedings*, LNCS, accepted for publication, 2016.

[40] A. Wijs and D. Bošnački. Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking. In *SPIN 2012, Proceedings*, volume 7385 of *LNCS*, pages 98–116, 2012.

[41] A. Wijs and D. Bošnački. GPUexplore : Many-Core On-the-Fly State Space Exploration Using GPUs. In *TACAS 2014, Proceedings*, volume 8413 of *LNCS*, pages 233–247, Grenoble, France, 2014.

[42] A. Wijs and D. Bošnački. Many-core on-the-fly model checking of safety properties using GPUs. *STTT*, 18(2):1–17, 2015.

[43] Z. Wu, Y. Liu, Y. Liang, and J. Sun. GPU Accelerated Counterexample Generation in LTL Model Checking. In *ICFEM 2014, Proceedings*, volume 8829 of *LNCS*, pages 413–429, Luxembourg, Luxembourg, 2014.

[44] Z. Wu, Y. Liu, J. Sun, J. Shi, and S. Qin. GPU Accelerated On-the-Fly Reachability Checking. In *ICECCS 2015, Proceedings*, pages 100–109, Gold Coast, Australia, 2015.