Partial-Order Reduction for GPU Model Checking

Thomas Neele[1]*, Anton Wijs[2], Dragan Bošnački[2], and Jaco van de Pol[1]

[1] University of Twente, Enschede, The Netherlands
[2] Eindhoven University of Technology, Eindhoven, The Netherlands

Abstract. Model checking using GPUs has seen increased popularity over the last years. Because GPUs have a limited amount of memory, only small to medium-sized systems can be verified. For on-the-fly explicit-state model checking, we improve memory efficiency by applying partial-order reduction. We propose novel parallel algorithms for three practical approaches to partial-order reduction. Correctness of the algorithms is proved using a new, weaker 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 amount of runtime overhead is acceptable.

1 Introduction

The practical applicability of model checking [1, 10] has often been limited by state-space explosion. Successful solutions to this problem have either depended on efficient algorithms for state space reduction, or on leveraging new hardware improvements. To capitalize on new highly parallel processor technology, multi-core [14] and GPU model checking [7] have been introduced. In recent years, this approach has gained popularity and multiple mainstream model checkers already have multi-threaded implementations [3, 9, 11, 14, 16]. In general, designing multi-threaded algorithms for modern parallel architectures brings forward new challenges typical for concurrent programming. For model checking, developing concurrent versions of existing state space algorithms is an important task.

The massive number of threads that run in parallel makes GPUs attractive for the computationally intensive task of state space exploration. Their parallel power can speed-up model checking by up to two orders of magnitude [2, 12, 26, 28]. Although the amount of memory available on GPUs has increased significantly over the last years, it is still a limiting factor.

In this work we aim to improve the memory efficiency of GPU-based model checking. Therefore, we focus on reconciling *partial-order reduction* (POR) techniques [13, 21, 23] with a GPU-based model checking algorithm [27]. POR exploits the fact that the state space may contain several paths that are similar, in the sense that their differences are not relevant for the property under consideration. By pruning certain transitions, the size of the state space can be reduced. Hence, POR has the potential to increase the practical applicability of GPUs in model checking.

* We gratefully acknowledge the support of NVIDIA Corporation with the donation of the GeForce Titan X used for this research.

Contributions We extend GPUEXPLORE [27], 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 [15], cample [6] and stubborn sets [23]. We improve the cample-set approach by computing clusters on-the-fly. Although our algorithms contain little synchronization, we prove that they satisfy the action ignoring proviso by introducing a new version of the so called cycle proviso, which is weaker than previous versions [21, 8], possibly leading to better reductions. Our implementation is evaluated by running benchmarks with models from several other tools. We compare the performance of each of the approaches with LTSMIN [16], which implements state-of-the-art algorithms for explicit-state multi-core POR.

The rest of the paper is organized as follows: section 2 gives an overview of related work and section 3 introduces the theoretic background of partial-order reduction and the GPU architecture. The design of our algorithms is described in section 4 and a formal correctness proof is given in section 5. Finally, section 6 presents the results obtained from executing our implementation on several models and section 7 provides a conclusion and suggestions for future work.

2 Related Work

Partial-order reduction. Bošnački et al. have defined cycle provisos for general state expanding algorithms [8] (GSEA, a generalization of *depth-first search* (DFS) and *breadth-first search* (BFS)). Although the proposed algorithms are not multi-core, the theory is relevant for our design, since our GPU model checker uses a BFS-like exploration algorithm.

POR has been implemented in several multi-core tools: Holzmann and Bošnački [14] implemented a multi-core version of SPIN that supports POR. They use a slightly adapted cycle proviso that uses information on the local DFS stack.

Barnat et al. [4] have defined a parallel cycle proviso that is based on a topological sorting of the state space. 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 provides competitive reductions. However, it is not clear from the paper whether it is slower or faster than a standard DFS-based implementation.

Laarman and Wijs [19] designed a multi-core version of POR that yields better reductions than SPIN's implementation, but has higher runtimes. The scalability of the algorithm is good up to at least 64 cores.

GPU model checking. General purpose GPU (GPGPU) techniques have already been applied in model checking by several people, all with a different approach: Edelkamp and Sulewski [12] perform successor generation on the GPU and apply delayed duplicate detection to store the generated states in main memory. Their implementation performs better than DIVINE, it is faster and consumes less memory per state. The performance is worse than multi-core SPIN, however.

Barnat et al. [2] perform state-space generation on the CPU, but offload the detection of cycles to the GPU. 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.

GPUEXPLORE by Wijs and Bošnački [26, 27] performs state-space exploration completely on the GPU. The 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. [5] have extended SPIN with a CUDA implementation. Their implementation has a significant overhead for smaller models, but performs reasonably well for medium-sized state spaces.

Wu et al. [28] also implemented a complete model checker in CUDA. They adopted several techniques from GPUEXPLORE, and added dynamic parallelism and global variables. The speed up gained from dynamic parallelism proved to be minimal. A comparison with a sequential CPU implementation shows a good speed-up, but it is not clear from the paper how the performance compares with other parallel tools.

GPUs have also been applied in probabilistic model checking: Bošnački et al. [7, 25] speed up value-iteration for probabilistic properties by solving linear equation systems on the GPU. Češka et al [9] implemented parameter synthesis for parametrized continuous time Markov chains.

3 Background

Before we introduce the theory of POR, we first establish the basic definitions of labelled transitions systems and concurrent processes.

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

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

Let $enabled(s) = \{\alpha | (s, \alpha, t) \in \tau\}$ be the set of actions that is enabled in state s and $succ(s, \alpha) = \{t | (s, \alpha, t) \in \tau\}$ the set of successors reachable through some action α. Additionally, we lift these definitions to take a set of states or actions as argument. The second argument of $succ$ is omitted when all actions are considered: $succ(s) = succ(s, A)$. If $(s, \alpha, t) \in \tau$, then we write $s \xrightarrow{\alpha} t$. We call a sequence of actions and states $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_n} s_n$ an *execution*. We call the sequence of states visited in an execution a *path*: $\pi = s_0 \dots s_n$. If there exists a path $s_0 \dots s_n$, then we say that s_n is *reachable* from s_0.

To specify concurrent systems consisting of a finite number of finite-state processes, we define a *network* of LTSs [20]. 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 $\Pi[1], \dots, \Pi[n]$ that are modelled as LTSs.*
- *V is a set of synchronization rules $(\boldsymbol{t}, a)$, where a is an action and $\boldsymbol{t} \in \{0, 1\}^n$ is a synchronization vector that denotes which processes synchronize on a.*

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.*
- *$\tau = \{(\langle s_1, \ldots, s_n\rangle, a, \langle t_1, \ldots, t_n\rangle) \,|\, \exists(\boldsymbol{t}, a) \in V : \forall i \in \{1..n\} : \boldsymbol{t}(i) = 1 \Rightarrow (s_i, a, t_i) \in \tau[i] \wedge \boldsymbol{t}(i) = 0 \Rightarrow s_i = t_i\}$ is the transition relation that follows from each of the processes and the synchronization rules.*
- *$\hat{s} = \langle \hat{s}[0], \ldots, \hat{s}[n]\rangle$ is the combination of the initial states of the processes.*

We distinguish two types of actions: (1) *local actions* that do not synchronize with other processes, i.e. all rules for those actions have exactly one element set to 1, and (2) *synchronizing actions* that do synchronize with other processes. In the rest of this paper we assume that local actions are never blocked, i.e. if there is a local action $\alpha \in A[i]$ then there is a rule $(\boldsymbol{t}, \alpha) \in V$ such that element i of $\boldsymbol{t}$ is 1 and the other elements are 0. Note that although processes can only synchronize on actions with the same name, this does not limit the expressiveness. Any network can be transformed into a network that follows our definition by proper action renaming.

During state-space exploration, we exhaustively generate all reachable states in $\mathcal{T}_{\mathcal{N}}$, starting from the initial state. When all successors of s have been identified, we say that s has been *explored*, and once a state s has been generated, we say that it is *visited*.

3.1 Partial-Order Reduction

We first introduce the general concept of a reduction function and a reduced state space.

Definition 4. *A reduced LTS can be defined according to some reduction function $r : S \rightarrow 2^A$. The reduction of $\mathcal{T}$ w.r.t. 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.*

POR is a form of state-space reduction for which the reduction function is usually computed while exploring the original state space (*on-the-fly*). That way, we avoid having to construct the entire state space and we are less likely to encounter memory limitations. However, a drawback is that we never obtain an overview of the state space and the reduction function might be larger than necessary.

The main idea behind POR is that not all interleavings of actions of parallel processes are relevant to the property under consideration. It suffices to check only one representative execution from each equivalence class of executions. To reason about this, we define when actions are *independent*.

Definition 5. *Two actions α, β are independent in state s if and only if $\alpha, \beta \in enabled(s)$ implies:*

- *$\alpha \in enabled(succ(s, \beta))$*
- *$\beta \in enabled(succ(s, \alpha))$*
- *$succ(succ(s, \alpha), \beta) = succ(succ(s, \beta), \alpha)$*

Actions are globally independent if they are independent in every state $s \in S$.

Based on the theory of independent actions, the following restrictions on the reduction function have been developed [10]:

C0a $r(s) \subseteq enabled(s)$.
C0b $r(s) = \emptyset \Leftrightarrow enabled(s) = \emptyset$.
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 $r(s)$.

C0b makes sure that the reduction does not introduce new deadlocks. C1 implies that all $\alpha \in r(s)$ are independent of $enabled(s) \setminus r(s)$. Informally, this means that only the execution of independent actions can be postponed to a later state. 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 section 4.

If r is a persistent set, then all deadlocks in an LTS $\mathcal{T}$ are preserved in $\mathcal{T}_r$. Therefore, persistent sets can be used to speed up checking for deadlocks. However, safety properties are generally not preserved due to the *action-ignoring problem*. This occurs whenever some action in the original system is ignored indefinitely, i.e. it is never selected for the reduction function. Since we are dealing with finite state spaces and condition C0b is satisfied, this can only occur on a cycle. To prevent action-ignoring, another condition, called the *action-ignoring proviso*, is applied to the reduction function.

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

Applying this proviso directly by means of Valmari's SCC approach [22] introduces quite some runtime overhead. For this reason, several stronger versions of the action-ignoring proviso have been defined, generally called *cycle provisos*. Since GPUEXPLORE does not follow a strict BFS order, we will use the *closed-set proviso* [8] (*Closed* is the set of states that have been visited and for which exploration has at least started):

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

3.2 GPU Architecture

CUDA[1] is a programming interface developed by NVIDIA to enable general purpose programming on a GPU. 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++.

On the hardware level, a GPU is divided up into several *streaming multiprocessors* (SM) that contain hundreds of cores. On the side of the programmer, threads are grouped into *blocks*. The GPU schedules thread blocks on the SMs. One SM can run multiple blocks at the same time, but one block cannot execute on more than one SM.

[1] https://developer.nvidia.com/cuda-zone

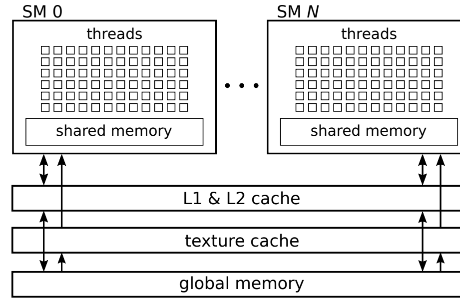

Fig. 1. Schematic overview of the GPU hardware architecture

Internally, blocks are executed as one or more *warps*. A warp is a group of 32 threads that move in lock-step through the program instructions.

Another important aspect of the GPU architecture is the memory hierarchy. Firstly, each block is allocated *shared memory* that is shared between its threads. 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 in between: the L1, L2 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 be accessed by the CPU ('host'), thus it also serves as an interface between the host and the device. Figure 1 gives a schematic overview of the architecture.

The bandwidth between the SMs and the global memory is used optimally when a continuous block of 32 integers is fetched by a warp. In that case, the memory transaction is performed in parallel. This is called *coalesced* access.

4 Design and implementation

4.1 Existing Design

GPUEXPLORE [27] is an explicit-state 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 global memory of the GPU is occupied by a large hash table that uses open addressing with rehashing. The hash table stores all the visited states, distinguishing the states that still need to be explored (*Open* set) from those that do not require this (*Closed*). It supports a *findOrPut* operation that inserts states if they are not already present. The implementation of findOrPut is thread-safe and lockless. It uses the *compareAndSwap* (CAS) operation to perform atomic inserts.

The threads are organized as follows: each thread is primarily part of a block. As detailed in section 3.2, the hardware enforces that threads are grouped in warps of size 32. We also created logical groups, called *vector groups*. The number of threads in a vector group is equal to the number of processes in the network (cf. section 3). When computing successors, threads cooperate within their vector group. Each thread has a

Algorithm 1: 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);
```

vector group thread id (*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*. When accessing global memory, threads cooperate within their warp and read continuous blocks of 32 integers for coalesced access. 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.

A high-level view on the algorithm of GPUEXPLORE is presented in Algorithm 1. This kernel is executed repetitively until all reachable states have been explored. Several iterations may be performed during each launch of the kernel (NUMITERATIONS is fixed 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 other 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, these states 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 successor computation and the hash table, we refer to [27].

In the following sections, we will show how the generation of successors on lines 7 and 8 can be adjusted to apply POR.

4.2 Ample-set Approach

The ample-set approach is based on the idea of *safe* actions [15]: an action is safe if it is independent of all actions of all other processes. While exploring a state s, if there

Algorithm 2: Successor generation under the ample-set approach

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

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 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 2. First, the successors of processes that have only local actions enabled are generated. These states are stored in the cache (line 4) 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 5). Then, line 8 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 9), then the cycle proviso has been satisfied and thread i reports it can apply reduction through the *reduceProc* shared variable (line 10). 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 18). In case no thread can apply reduction, the algorithm continues as normal (line 20).

4.3 Clustered Ample-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 [6]. 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 [6] determine a tree-shaped cluster hierarchy a priori and by hand, our implementation computes the cluster on-the-fly. 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 [6], it is argued that computing the cluster on-the-fly is an expensive operation, so it should be avoided. Our approach is, when we are exploring a state s, 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 algorithm, the reasoning of [6] still applies.

The algorithm for computing cample-sets 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 Algorithm 2. Dynamic allocation in shared memory is not supported by CUDA. The only alternative is to alternate between successor generation and checking whether the last state is marked as *new* in global memory. During this process, each thread tracks whether the generated successors satisfy the cycle proviso and with which other processes it synchronizes, based on the synchronization rules. The next step is to share this information via shared memory. Then, each thread computes a fixed-point as detailed above. The group leader selects the smallest of those fixed-points as cluster. All actions of processes in that closure will form the cample set. Finally, states are marked as *old* or *new* depending on whether they follow from an action in the cample set.

4.4 Stubborn-set Approach

The stubborn-set approach was originally introduced by Valmari [23] and can yield better reductions than the ample-set approach. This technique is more complicated and

can lead to overhead, since it reasons about all actions, even those that are disabled. The algorithm starts by selecting one enabled action and builds a stubborn set by iteratively adding actions as follows: for enabled actions α, all actions that are dependent on α are added. For disabled actions β, all actions that can enable β 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. One bitvector can be used to represent a subset of the synchronization rules and the local actions. In case we apply the cycle proviso, we need four such bitvectors: to store the stubborn set, the set of enabled actions, the set of actions that satisfy the cycle proviso and a work set to track which actions still need to be processed. 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 approach that results in an acceptable computational overhead.

To reduce the size of the computed stubborn set, we use the *necessary disabling sets* and the heuristic function from Laarman et al. [17]. Contrary to their implementation, we do not compute a stubborn set for all possible choices of initial action. Our implementation deterministically picks an action, giving preference to local actions. Effectively, we sacrifice some reduction potential in order to minimize the overhead of computing a stubborn set.

In GPUEXPLORE, it is not possible to determine in constant time whether a certain action is enabled. Therefore, we chose to generate the set of enabled actions before computing the stubborn set. This also allows us to check which actions satisfy the cycle proviso. With this information saved in shared memory, a stubborn set can be computed efficiently. In case the set of actions satisfying the cycle proviso is empty, the set of all actions is returned. Otherwise, the group leader selects one initial action that satisfies the cycle proviso for the work set. Then, all threads in the group execute the closure algorithm in parallel. After computation of the stubborn set has finished, all successors following from actions in the set are generated and stored in the cache.

5 Proof of Correctness

The correctness of applying Bošnački et al.'s [8] closed-set proviso C2c in a multi-threaded environment is not immediately clear. The original correctness proof is based on the fact that for every execution, states are removed from *Open* (the set of unexplored states) in a certain sequence. In a multi-threaded algorithm, however, two states may be removed from *Open* at the same time. To prove that the algorithms introduced in the previous section satisfy the action ignoring proviso, we introduce a new version of the cycle proviso:

Lemma 1. (closed-set cycle proviso) *If a reduction algorithm satisfies conditions C0a, C0b and C1 and selects for every cycle* $s_0 \xrightarrow{\alpha_0} s_1 \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_{n-1}} s_n \xrightarrow{\alpha_n} s_0$ *in the reduced state space with* $\beta \in \mathit{enabled}(s_0)$ *and* $\beta \neq \alpha_i$ *for all* $0 \leq i \leq n$, (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.*

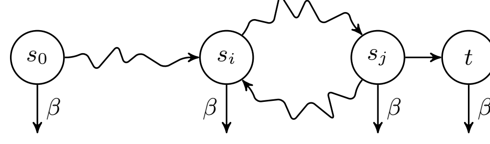

Fig. 2. 'Lasso' shaped path from the proof of Lemma 1

Proof. Suppose that action $\beta \in \mathit{enabled}(s_0)$ 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 2). 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. 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. □

Before we prove that our algorithms satisfy the action ignoring proviso, it is important to note three things. Firstly, that the work gathering function on line 4 of Algorithm 1 moves the gathered states from *Open* to *Closed*. Secondly, the ample/stubborn set generated by our algorithms satisfies conditions C0a, C0b and C1, also when executed by multiple vector groups (the proof for this is omitted from this paper). And lastly, in this theorem the ample-set approach is used as an example, but the reasoning applies to all three algorithms.

Theorem 1. *Algorithm 2 produces a persistent set that satisfies our action-ignoring proviso, even when executed 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 being ignored. 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 1). There are two possibilities regarding the processing of state s_{i-1}:

- s_{i-1} is gathered from *Open* at exactly the same time as s_i. When the processing for s_{i-1} arrives at line 9 of Algorithm 2, s_i will be in *Closed*.

Table 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	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

model	#states	#transitions	stub. set size
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
lamport8	62,669,317	304,202,665	305
szymanski5	79,518,740	922,428,824	481
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

– s_{i-1} 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 the closed-set cycle proviso (Lemma 1). □

6 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]. Since GPUEXPLORE only accepts EXP models as input, we added an EXP language front-end to LTSMIN [16] 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 and tools. Moreover, speed comparisons have already been done before [5, 27, 28].

GPUEXPLORE was benchmarked on an NVIDIA Titan X, which has 24 SMs and 12GB of global memory. We allocated 5GB for the hash table. Our code was run on 3120 blocks of 512 threads and performed 10 iterations per kernel launch (cf. section 4.1), since these numbers give the best performance [27].

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 described by Laarman et al. [17].

The models that were used as benchmarks have different origins. `Cache`, `sieve`, `odp`, `transit` and `asyn3` are all EXP models from the examples included in the CADP toolkit[3]. `1394`, `acs` and `wafer stepper` are originally mCRL2[4] models and have been translated to EXP. The `leader_election`, `lamport`, `lann`, `peterson` and `szymanski` models come from the BEEM database and have been translated from

[2] Sources are available at https://github.com/ThomasNeele/GPUexplore

[3] http://cadp.inria.fr

[4] http://mcrl2.org

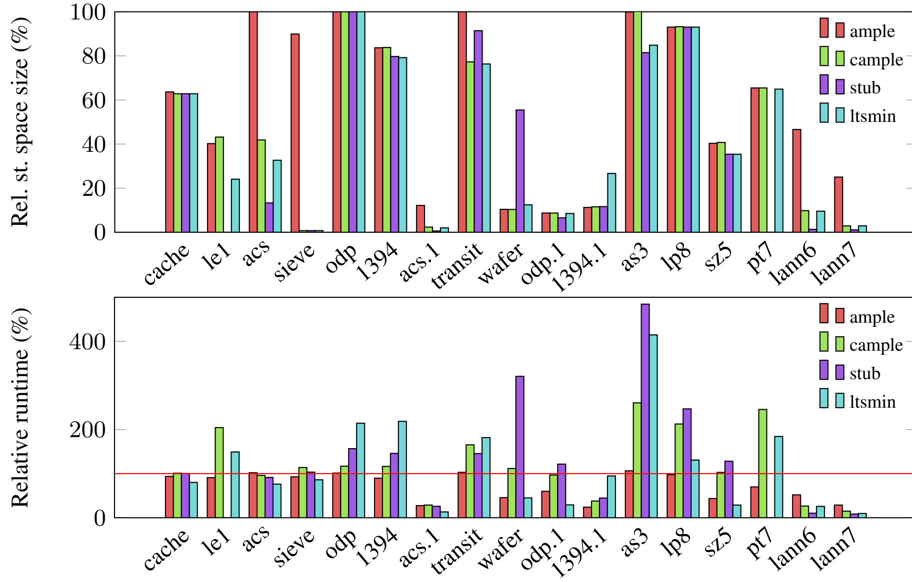

Fig. 3. State space and runtime of POR (no cycle proviso) relative to full exploration.

DVE to EXP. The models with a .1-suffix are enlarged versions of the original models [27]. The details of the models can be found in Table 1. 'stub. set size' indicates the maximum size of the stubborn set, which is equal to the amount of synchronization rules plus the total amount of local actions.

For the first set of experiments, 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 ten times. The average size of the reduced state space relative to the full state space is plotted in the first chart of Figure 3 (the full state space has a size of 100% for each model). The error margins are not depicted because they are very small (less than one percent point).

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 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* [18].

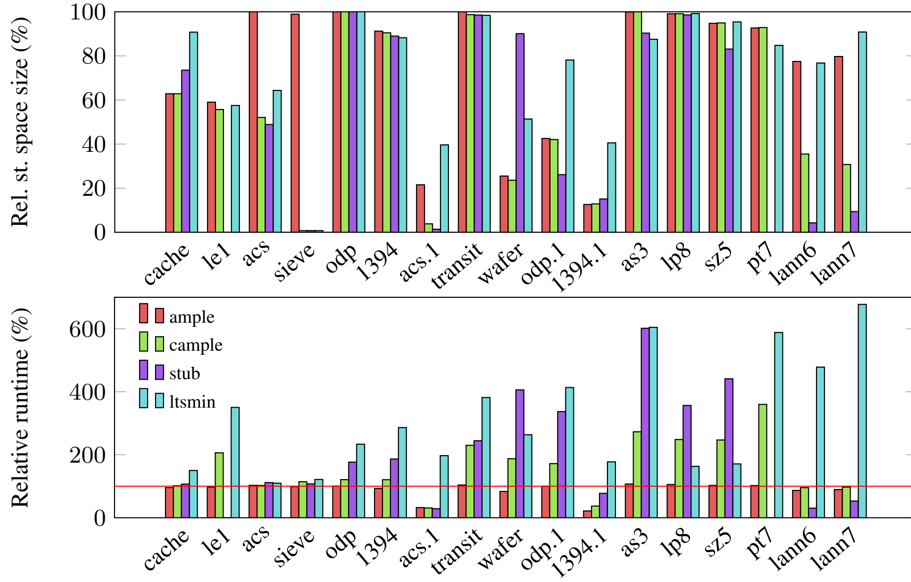

Fig. 4. State space and runtime of POR with cycle proviso relative to full exploration.

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 the second chart of Figure 3. 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 very 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, both the cample and stubborn-set approach suffer from significant overhead. When no or little reduction is possible, this slows down the exploration process by 2.6 times and 4.8 times respectively for the `asyn3` model. This model has the largest amount of synchronization rules after the `leader_election1` and `peterson7` models.

For the smaller models, the speed-up that can be gained by the parallel power of thousands of threads is limited. If a *frontier* (search layer) of states is smaller than the amount of states that can be processed in parallel, then not all threads are occupied and the efficiency drops. This problem can only get worse under POR. For the largest models, the overhead for LTSMIN is 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 set of experiments, we used POR with cycle proviso. Figure 4 shows the size of the state space and the runtime. As expected, less reduction is achieved. The checking of the cycle proviso induces only a little extra overhead (not more than 5%) for the ample-set and the cample-set approach. The extra overhead for the stubborn-set

Table 2. Average relative size of reduced state spaces

average size $\mathcal{T}_r$ (%)	ample	cample	stubborn	ltsmin
no proviso	58.97	43.08	42.30	41.80
cycle proviso	73.74	56.49	55.26	73.45

approach can be significant, however: up to 36% for the `lamport8` model (comparing the amount of states visited per second). Here, 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. Our approach, where the set consisting of only the initial action already satisfies the cycle proviso, often finds a smaller stubborn set. Therefore, GPUEXPLORE achieves a higher amount of reduction when applying the cycle proviso.

Table 2 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.

7 Conclusion

We have shown that partial-order reduction for many-core platforms has similar or better reduction potential than for multi-core platforms. Although the implementation suffers from overhead due to the limitations on shared memory, it increases the memory efficiency and practical applicability of GPU model checking. When the cycle proviso is applied, our approach performs better than LTSmin.

The cample-set approach performs best with respect to our goal of saving memory with limited runtime overhead. With our improvement of dynamic clusters, it often achieves the same reduction as the stubborn-set approach. Additionally, it can also be applied to models with a large amount of local actions and synchronization rules.

Further research into the memory limitations of GPU model checking is necessary. A possible approach is to implement a multi-GPU version of GPUEXPLORE. Another direction for future work is to support POR for linear-time properties, as recently, GPUEXPLORE was extended to check such properties on-the-fly [24].

Acknowledgements The authors would like to thank Alfons Laarman for his suggestions on how to improve this work.

References

1. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
2. Barnat, J., Bauch, P., Brim, L., Češka, M.: Designing fast LTL model checking algorithms for many-core GPUs. Journal of Parallel and Distributed Computing 72(9), 1083–1097 (2012)
3. Barnat, J., Brim, L., Ročkai, P.: DiVinE multi-core - A parallel LTL model-checker. In: ATVA. LNCS, vol. 5311, pp. 234–239. Springer (2008)
4. Barnat, J., Brim, L., Ročkai, P.: Parallel partial order reduction with topological sort proviso. In: Proc. of the 8th IEEE International Conference on Software Engineering and Formal Methods. pp. 222–231. IEEE (2010)

5. Bartocci, E., Defrancisco, R., Smolka, S.A.: Towards a GPGPU-Parallel SPIN Model Checker. In: SPIN 2014, Proceedings. pp. 87–96. ACM, San Jose, CA, USA (2014)
6. Basten, T., Bošnački, D., Geilen, M.: Cluster-Based Partial-Order Reduction. ASE 2004, Proceedings 11(4), 365–402 (2004)
7. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: Parallel probabilistic model checking on general purpose graphics processors. STTT 13(1), 21–35 (2010)
8. Bošnački, D., Leue, S., Lluch-Lafuente, A.: Partial-order reduction for general state exploring algorithms. STTT 11(1), 39–51 (2009)
9. Češka, M., Pilař, P., Paoletti, N., Brim, L., Kwiatkowska, M.: PRISM-PSY: Precise GPU-Accelerated Parameter Synthesis for Stochastic Systems. In: TACAS. LNCS, vol. 9636, pp. 367–384. Springer (2016)
10. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (2001)
11. Dalsgaard, A.E., Laarman, A., Larsen, K.G., Olesen, M.C., Van De Pol, J.: Multi-core reachability for timed automata. In: FORMATS. LNCS, vol. 7595, pp. 91–106. Springer (2012)
12. Edelkamp, S., Sulewski, D.: Efficient explicit-state model checking on general purpose graphics processors. In: SPIN. LNCS, vol. 6349, pp. 106–123. Springer (2010)
13. Godefroid, P., Wolper, P.: A Partial Approach to Model Checking. Information and Computation 110(2), 305–326 (1994)
14. Holzmann, G.J., Bošnački, D.: The design of a multicore extension of the SPIN model checker. IEEE Transactions on Software Engineering 33(10), 659–674 (2007)
15. Holzmann, G.J., Peled, D.: An Improvement in Formal Verification. In: IFIP Advances in Information and Communication Technology. pp. 197–211. Springer (1995)
16. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: High Performance Language-Independent Model Checking. In: TACAS. LNCS, vol. 9035, pp. 692–707. Springer (2015)
17. Laarman, A., Pater, E., van de Pol, J., Weber, M.: Guard-Based Partial-Order Reduction. In: SPIN. LNCS, vol. 7976, pp. 227–245. Springer (2013)
18. Laarman, A., van de Pol, J., Weber, M.: Parallel Recursive State Compression for Free. In: SPIN. LNCS, vol. 6823, pp. 38–56. Springer (2011)
19. Laarman, A., Wijs, A.: Partial-Order Reduction for Multi-core LTL Model Checking. In: HVC. LNCS, vol. 8855, pp. 267–283. Springer (2014)
20. Lang, F.: Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-The-Fly Verification Methods. In: IFM. LNCS, vol. 3771, pp. 70–88. Springer (2005)
21. Peled, D.: All from One, One for All: on Model Checking Using Representatives. In: CAV. LNCS, vol. 697, pp. 409–423. Springer (1993)
22. Valmari, A.: A Stubborn Attack on State Explosion. In: CAV. LNCS, vol. 531, pp. 156–165. Springer (1991)
23. Valmari, A.: Stubborn sets for reduced state space generation. In: Advances in Petri Nets. vol. 483, pp. 491–515 (1991)
24. Wijs, A.: BFS-Based Model Checking of Linear-Time Properties With An Application on GPUs. In: CAV. LNCS, accepted for publication (2016)
25. Wijs, A., Bošnački, D.: Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking. In: SPIN. LNCS, vol. 7385, pp. 98–116. Springer (2012)
26. Wijs, A., Bošnački, D.: GPUexplore : Many-Core On-the-Fly State Space Exploration Using GPUs. In: TACAS. LNCS, vol. 8413, pp. 233–247 (2014)
27. Wijs, A., Bošnački, D.: Many-core on-the-fly model checking of safety properties using GPUs. STTT 18(2), 1–17 (2015)
28. Wu, Z., Liu, Y., Sun, J., Shi, J., Qin, S.: GPU Accelerated On-the-Fly Reachability Checking. In: Proc. of the 20th International Conference on Engineering of Complex Computer Systems. pp. 100–109. IEEE (2015)