Formal Methods in System Design manuscript No.
(will be inserted by the editor)

Efficient GPU Algorithms for Parallel Decomposition of Graphs into Strongly Connected and Maximal End Components

Anton Wijs · Joost-Pieter Katoen · Dragan Bošnački

Received: date / Accepted: date

Abstract This article presents parallel algorithms for component decomposition of graph structures on General Purpose Graphics Processing Units (GPUs). In particular, we consider the problem of decomposing sparse graphs into strongly connected components, and decomposing stochastic games (such as Markov decision processes) into maximal end components. These problems are key ingredients of many (probabilistic) model-checking algorithms. We explain the main rationales behind our GPU-algorithms, and show a significant speed-up over the sequential (as well as existing parallel) counterparts in several case studies.

Keywords parallel graph algorithms · strongly connected components · maximal end components · probabilistic model checking · Markov decision processes · GPU

1 Introduction

Strongly connected components (SCCs, for short) are sub-graphs in which each pair of states is mutually reachable. Finding maximal SCCs, i.e., SCCs that are not contained in others, is a key ingredient of various model-checking algorithms. To mention a few, this applies to the standard verification algorithms for CTL-formulas of the form $\mathsf{EG}\,\varphi$ and for verifying fair CTL [3, Ch. 6] where so-called fair

This work was sponsored by the NWO Exacte Wetenschappen, EW (NWO Physical Sciences Division) for the use of supercomputer facilities, with financial support from the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (Netherlands Organisation for Scientific Research, NWO), as well as the EU MEALS project and the EU FP7 CARP project.

Anton Wijs
Eindhoven University of Technology, The Netherlands
E-mail: A.J.Wijs@tue.nl

Joost-Pieter Katoen
RWTH Aachen University, Germany
E-mail: Katoen@cs.rwth-aachen.de

Dragan Bošnački
Eindhoven University of Technology, The Netherlands
E-mail: D.Bosnacki@tue.nl

SCCs play an important role. Checking language emptiness [32] is based on SCCs, and efficient model-checking algorithms for discrete-time Markov chains exploit SCC decomposition [1]. The high relevance of SCCs has led to various dedicated variants of Tarjan's classical algorithm [30] such as a symbolic variant [8] and a plethora of parallel [6, 22, 25] algorithms. In the context of probabilistic model checking, a generalisation of SCCs – known as maximal end components (MECs) – play a pivotal role [16, 2]. Determining MECs is a main step in the verification of qualitative and quantitative properties on Markov decision processes (MDPs) and continuous-time variants thereof. MDPs are an important class of models used for the analysis of probabilistic systems consisting of several components running in parallel. Parallelism is modelled by non-determinism whereas the steps within a component may be probabilistic (e.g., modelling a coin flip). MDP model checking is a very active branch of probabilistic model checking with applications in amongst others planning and randomised distributed algorithms. MECs are maximal strongly connected sub-graphs in which the MDP can ensure to reside when playing against a probabilistic adversary. MEC decomposition of MDPs is a preprocessing step of probabilistic model checking to determining almost-sure limiting properties [3, Ch. 10] such as almost-sure repeated reachability and limiting Rabin acceptance conditions. They are also relevant for checking ω-regular properties on MDPs under fairness constraints [3, Ch. 10]. Other applications of MEC decomposition include the analysis of multi-player stochastic games [31], recent approaches to combined worst-case and expected value objectives for mean pay-off games [11], as well as incremental verification techniques for MDPs [24]. Algorithms for MEC decomposition are still an active area of research. Improvements of the traditional sequential algorithms for determining MECs [3, 16, 2] have been reported [12] and were tailored to MDPs with low tree-width [15]. Recently, also a first dynamic algorithm has been given that maintains the MEC decomposition of a graph under a sequence of edge insertions or deletions [14].

In this article, we provide new algorithms to efficiently decompose graphs into SCCs and MECs by exploiting GPUs (Graphics Processing Units). Our decomposition algorithms build upon three key principles. First, inspired by the Forward-Backward algorithm (FB) [19], each thread combines *a forward and a backward reachability search* so as to identify SCCs. Previous work on GPU-based SCC decomposition [6, 22, 25] identified the FB algorithm (combined with a trimming procedure to remove trivial SCCs) as the best performing one for general input graphs. Opposed to these works, we focus on graphs that are commonly observed in model checking, i.e., sparse graphs with a low average out-degree (number of outgoing transitions per state) and tailor our algorithms to treat these graphs efficiently. The backward and forward search are started from some common state, called the pivot.

The second main principle is to exploit a *novel pivot selection* strategy which turns out to be simple and efficient. It enforces race conditions between threads to pseudo-randomly select pivots. Compared to our earlier work [36], this strategy has been further optimised in two ways: first of all, we use so-called *warp-aggregation* to reduce the number of racing threads, and second of all, when used for MEC decomposition, an SCC decomposition procedure following a MEC decomposition iteration can exploit the results obtained so far (as inspired by [12]) to further restrict the number of racing threads.

Finally, we optimise the memory management to achieve *coalesced memory access* by the individual threads, i.e., data access can be accomplished in a single memory fetch. Altogether this alleviates memory latency and thread divergence where part of the threads execute one branch of the common code, while others take another branch.

The overall memory requirements are significantly lower than for competitive algorithms [6] as besides the input graph $G = (V, E)$, only a single additional integer array of size $|V|$ is needed to store decomposition results. Given the restricted memory size on a GPU, this memory reduction is essential. Our GPU-based MEC decomposition algorithm uses the same principles as the SCC algorithm; it can be viewed as a parallel version of the standard sequential algorithms [3, 16, 2]. To the best of our knowledge, this is the first GPU-based MEC decomposition procedure. We implemented our algorithms using CUDA[1] for NVIDIA GPUs, and ran them on examples of the PRISM benchmark suite [23]. Speed-up factors of 15-30 and 79 have been achieved for SCC and MEC decomposition, respectively. For SCC decomposition, this is a significant improvement over previous results (e.g. [6]) for sparse graphs with a low average out-degree.

Exploiting general purpose GPUs (GPGPUs, for short) in the setting of model checking is not new. Thanks to efforts of several research groups [10, 5, 18], GPGPUs have been applied to significantly improve the run times of model checking algorithms. In the context of probabilistic model checking, these improvements usually targeted the numerical part of the algorithms, so as to exploit the inherent advantages of the GPUs [10, 9, 34]. More recently, we presented an on-the-fly search algorithm for standard model checking running entirely on GPUs [35], and how to perform strong and branching bisimilarity checking on GPUs [33].

This article is based on the conference version [36], extending it in the following ways:

1. In the preliminary section:
   (a) We provide the proofs of all theoretical results.
   (b) A running example has been added to illustrate the presented notions.
   (c) Besides the FB algorithm with trimming, we also discuss an alternative version proposed by Bloem *et al.* [7], which we refer to as BFBT, that offers an advantage over standard FB in terms of overall complexity, but also a disadvantage regarding parallelism. Like FB, we also consider SCC and MEC decomposition based on BFBT.
   (d) A section has been added in which we reason how the basic algorithms can be adapted for parallel execution.
2. Multiple pseudo-code descriptions have been added, and the accompanying text explains in more detail how the SCC and MEC decomposition procedures work.
3. We propose two new optimisations regarding the so-called *pivot selection* procedure, which is a key step in our GPU decomposition algorithms.
4. Finally, the experimental section has been extended to validate the new amendments of the pivot selection and the use of BFBT.

Organisation of the article. Section 2 treats the basics of MDPs, MECs and relevant SCC and MEC decomposition algorithms. Section 3 gives an introduction to

[1] `http://www.nvidia.com/object/cuda_home_new.html`.

the typical architecture of a GPU and the various important aspects and notions relevant for GPU programming and understanding the article. In Section 4, we discuss related work, specifically focussing on how to perform Breadth-First Search (BFS) efficiently on GPUs. Then, in Section 5, the various GPU procedures that we have developed to efficiently perform SCC decomposition on GPUs are presented, followed by a discussion in Section 6 how this approach can be extended to achieve MEC decomposition. Finally, Section 7 presents the experimental results, and Section 8 concludes.

2 Preliminaries

This section gives an introduction to the main concepts of MDPs and MECs [3, Ch. 10], presents the parallel FB algorithm for SCC decomposition [19] and the standard sequential algorithm for MEC decomposition [16, 2] of MDPs.

2.1 Markov Decision Processes and Maximal-End Components

Let $\Delta(X)$ denote the set of probability distributions over the countable set X, i.e., the set of functions $\mu : X \to [0, 1]$ with $\sum_{x\in X} \mu(x) = 1$.

Definition 1 (Markov Decision Process) A *Markov decision process (MDP)* is a tuple $M = (S, \hat{s}, T)$, where S is a finite set of *states*, $\hat{s} \in S$ is the *initial state*, and $T : S \to 2^{\Delta(S)}$ is the *transition function* with $T(s) \neq \emptyset$ and $T(s)$ is finite for all $s \in S$.

The transition function T maps every state $s \in S$ to a finite, non-empty set of distributions over S. In state s, one of the distributions in $\mu \in T(s)$ is selected non-deterministically, and the MDP evolves to state s' with probability $\mu(s')$. As $T(s)$ is non-empty for every state, this procedure can be repeated *ad infinitum.* For state s, $T(s)$ can be viewed as the set of distributions that are selected in a non-deterministic manner. Alternatively, an MDP can be considered as a single-player game in which the system plays against a random adversary. An MDP naturally induces a digraph in the following sense.

Example 1 Consider the MDP depicted in Figure 1 with $S = \{s_0, \ldots, s_7\}$, $\hat{s} = s_0$ and e.g., $T(s_0) = \{\alpha, \beta\}$ with $\alpha(s_1) = 1$, $\beta(s_2) = 1/3$, and $\beta(s_4) = 2/3$.

Definition 2 (MDP Graph) The *induced labelled digraph* of MDP $M = (S, \hat{s}, T)$ is $G = (V, E)$ with $V = S$ is the set of *vertices* and $E \subseteq V \times \Delta(V) \times V$ is the set of *labelled edges* defined by: $(u, \mu, v) \in E$ iff $\mu(v) > 0$ for $\mu \in T(u)$.

Intuitively speaking, there is a μ-labelled edge between two vertices (states) u and v whenever v is in the support of distribution μ in $T(u)$. For node u and distribution μ, let $E_\mu(u) = \{v \in V \mid (u, \mu, v) \in E\}$. We call $E_\mu(u)$ the set of *target vertices (states)* of the *source vertex (state)* u under distribution μ. Moreover, let $E(u) = \bigcup_\mu E_\mu(u)$. For labelled digraphs we adopt the standard graph-theoretical notions like paths, cycles, components, etc.. An MDP graph $G = (V, E)$ is strongly connected iff for every two vertices $u, v \in V$ there is a path from u to v and a path

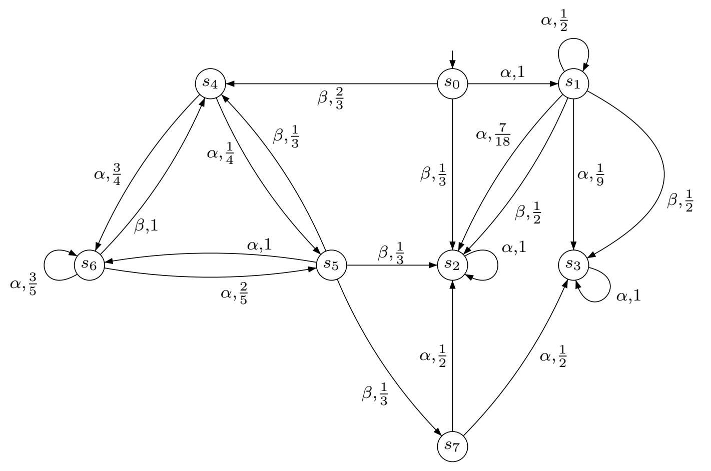

Fig. 1 Example MDP. The corresponding induced graph is obtained by omitting the numbers behind the actions in the transition labels.

from v to u. The set of nodes $C \subseteq V$ is a *strongly connected component* (SCC) of G iff G restricted to C, denoted $G{\upharpoonright}C$, i.e., the graph $G{\upharpoonright}C = (C, (C \times \Delta(C) \times C) \cap E)$, is strongly connected. SCC C is *maximal* iff there is no SCC $C' \neq C$ with $C \subset C'$. In the sequel, unless stated otherwise, we use the abbreviation SCC for maximal SCCs. In the following, let $G = (V, E)$ be an MDP graph.

Definition 3 (SCC Decomposition) An *SCC decomposition* of graph $G = (V, E)$ is a partitioning of V that consists of all maximal SCCs of G.

It is convenient to distinguish vertices that are potentially "closed" in the sense that for at least one non-deterministic choice (distribution) all transitions remain within a given set.

Definition 4 (E-Closed Nodes) Vertex $v \in V$ is *existentially closed* (e-closed) for $X \subseteq V$ iff $E_\mu(v) \subseteq X$ for some $\mu \in T(v)$.

Definition 5 (End-Components) $U \subseteq V$ is an *end-component* of MDP graph G if $G{\upharpoonright}U$ is strongly connected and every $u \in U$ is e-closed for U.

Strictly speaking, a set of nodes does not uniquely identify an end-component, but for each node one also has to keep track for which distribution μ the node is e-closed for the end-component at hand. As this does not play a role in the sequel, we refrain from these technical details. An end-component is a finite set of nodes such that for some choice of the distributions in all these nodes, the MDP will stay in these nodes with probability one. End-components thus play a similar role as terminal SCCs in digraphs. End-components that share common nodes can be merged into a single end-component. A *maximal* end-component (MEC) of G is an end-component C for which there is no end-component $C' \neq C$ such that $C \subset C'$. Observe that every vertex in V belongs to at most one maximal end-component.

Example 2 Node s_1 is e-closed for $X = \{s_0, s_1, s_2, s_3\}$ since $E_\alpha(s_1) = \{s_1, s_2, s_3\} \subseteq X$; evidently, node s_1 is not e-closed for $X' = \{s_1\}$ since $E_\alpha(s_1) \not\subseteq X'$ and $E_\beta(s_1) = \{s_2, s_3\} \not\subseteq X'$. For node s_6 we have $E_\alpha(s_6) = \{s_5, s_6\}$ and $E_\beta(s_6) = \{s_4\}$. Thus, $E(s_6) = \{s_4, s_5, s_6\}$.

Definition 6 (MEC Decomposition) A *MEC decomposition* of MDP graph G is the partitioning of V into the MECs of G and the set of vertices that do not belong to any MEC (of G).

For the description of the MDP algorithms (below) we define the notion of attractor. Stated in words, an attractor is a set of vertices in which the MDP may reside with positive probability no matter which distributions are non-deterministically selected.

Definition 7 (Attractor) The *attractor* $\mathrm{Attr}(U)$ of $U \subseteq V$ is defined as $\mathrm{Attr}(U) = \bigcup_{i \geq 0} U_i$ where U_i is defined inductively by:

– $U_0 = U$, and
– $U_{i+1} = U_i \cup \{u \in V \mid \forall \mu.\, E_\mu(u) \cap U_i \neq \emptyset\}$, for $i \geq 0$.

The attractor $\mathrm{Attr}(U)$ contains U plus all vertices from which (the vertices in) U can be reached via at least one transition regardless of the resolution of the non-deterministic choices by the adversary.

Example 3 Consider again the running example MDP. It follows that $\{s_4, s_5, s_6\}$ is a MEC, as well as $\{s_2\}$ and $\{s_3\}$. These are the only MECs in the running example. For $U = \{s_3\}$ we have $\mathrm{Attr}(U) = \{s_1, s_3\}$.

The MEC-decomposition algorithm discussed later on exploits the following two results from [13]. (The formulation of the lemmata and the corresponding proofs are adapted to our definition of MDP graphs.) The first result identifies some of the vertices that do not belong to any MEC and thus can be removed without affecting the MEC decomposition of the rest of the MDP graph.

Lemma 1 (Removing Attractor Nodes) *Let $G = (V, E)$ be an MDP graph.*

1. *For SCC C in G, let $U = \{v \in C \mid \forall \mu.\, E_\mu(v) \nsubseteq C\}$ and $Z = Attr(U) \cap C$. Then: for every MEC X of G it holds that $Z \cap X = \emptyset$.*
2. *Let C be a MEC in G and $Z = Attr(C) \setminus C$. Then: for every MEC X of G it holds that $Z \cap X = \emptyset$.*

Proof 1. Assume there is a MEC X with $Z \cap X \neq \emptyset$. We first show that $X \subseteq C$. From $Z \subseteq C$ and $Z \cap X \neq \emptyset$ it follows that $X \cap C \neq \emptyset$. Since X and C share nodes and are both strongly connected, it follows that they belong to the same SCC, which because of the maximality requirement is C itself, hence $X \subseteq C$. Now it suffices to show that $X \cap \mathrm{Attr}(U) = \emptyset$. This is done by contraposition. Using the structure of the attractor definition (Definition 7) we show by induction that $X \cap U_i = \emptyset$, for all i. The base case for $U_0 = U$ holds because $X \cap U \neq \emptyset$ would imply that X is not e-closed. For the inductive step, suppose that, for some k, $X \cap U_k = \emptyset$ and $X \cap U_{k+1} \neq \emptyset$. Let us consider a node $u \in X \cap U_{k+1}$. By Definition 7, we have $\forall \mu. E_\mu(u) \cap U_k \neq \emptyset$. But this is impossible since it contradicts the induction hypothesis. Therefore, $X \cap U_{k+1} = \emptyset$, which establishes the inductive step of the proof.

2. In case $X = C$ the claim follows directly from $Z = \mathrm{Attr}(C) \setminus C$. So, we assume $X \neq C$. The proof follows a similar inductive pattern like for the first item. We show that each set U_i in Definition 7 is disjoint from X. For the base step $U_0 = C$, since $X \neq C$ and X and C as SCCs are by definition disjoint, we have

Algorithm 1 FB with Trimming (FBT)

```
Require: graph $G = (V, E)$, set $J \subseteq V$
Ensure: SCC decomposition of $G$ is given
    $V' \leftarrow$ TRIM($V$) produces trivial SCCs
 2: if $V' \neq \emptyset$ then
    $pivot \leftarrow$ SELECTPIVOT($V' \cap J$)
 4: $F \leftarrow$ FWDBFS($pivot, (V', E)$)
    $B \leftarrow$ BWDBFS($pivot, (V', E)$)
 6: remove SCC $F \cap B$ from $V'$
    do in parallel
 8:     FBT($((F \setminus B), E), J$)
        FBT($((B \setminus F), E), J$)
10:     FBT($((V' \setminus (B \cup F)), E), J$)
```

$X \cap U_0 = \emptyset$. Suppose that, for some k, $X \cap U_k = \emptyset$ and $X \cap U_{k+1} \neq \emptyset$. As above, using Definition 7 we conclude that for some $u \in X \cap U_{k+1}$ $\forall \mu. E_\mu(u) \cap U_i \neq \emptyset$, which contradicts the fact that X, being a MEC, must be e-closed.

Example 4 Consider the MEC $C = \{s_3\}$ in the running example MDP. $Z = \text{Attr}(C) \setminus C = \{s_1, s_3\} \setminus \{s_3\} = \{s_1\}$, and indeed nodes s_0 and s_1 do not belong to any MEC of the MDP, as by Lemma 1.2.

The second result from [13] provides a sufficient criterion for an SCC to be a MEC. Lemma 2 below formally establishes the fact that every bottom SCC, i.e., an SCC C such that all transitions from C lead back to C, is a MEC.

Lemma 2 (Each bottom SCC is a MEC) *A bottom SCC C of the MDP graph $G = (V, E)$, i.e., an SCC C with $E(v) \subseteq C$ for all $v \in C$, is a MEC.*

Proof From the premise of the lemma we have that C is closed and consequently it is also e-closed. Since C is also an SCC, it follows that C is a MEC.

2.2 SCC Decomposition using Forward-Backward Search

Many algorithms exist to perform SCC decomposition. Linear-time algorithms such as the ones by Tarjan [30] and Dijkstra [17] are based on depth-first search and thus very hard to parallelize, especially when the goal is to run thousands of threads in parallel as is the case with GPUs. An alternative for SCC decomposition is the Forward-Backward algorithm (FB, for short) proposed by Fleischer *et al.* [19]. A similar approach was proposed by Xie and Beerel for symbolic model checking, but without noting the potential for parallel execution [37]. This algorithm is based on a breadth-first search (BFS) strategy, combining a forward and a backward search. It has worst-case complexity $O(|V|^2 + |V| \cdot |E|)$, but offers great potential for GPU-based parallelization.

The Forward-Backward algorithm of Fleischer *et al.* is presented in Algorithm 1, with two modifications: first of all, a so-called *trimming* procedure has been added at line 1, which is discussed later in this section. Because of this, we refer to this algorithm from now on as FBT. Second of all, besides a graph $G = (V, E)$, it also takes as input a *candidate set* of vertices $J \subseteq V$. The algorithm starts by (randomly) selecting a *pivot* vertex p (see Algorithm 1, line 3) from J.

The SCC to which p belongs is then found by performing both a *forward* BFS and a *backward* BFS starting from p, to determine the forward and backward closure (of p), respectively (Algorithm 1, lines 4-5). The intersection of the vertices reached via the forward and backward BFSs constitutes an SCC (and is removed, Algorithm 1, line 6). The graph vertices are then partitioned into the vertices belonging only to the forward closure, those only in the backward closure, and those outside both closures. These subsets are referred to as *search regions*. Subsequently, FBT can be invoked recursively in parallel on the three search regions. This can be done, since all other, not yet detected SCCs, are contained in one of these search regions.

A necessary condition for the correctness of the FBT algorithm is that set J does not become $\emptyset$ as long as at least one of the generated search regions is not empty, i.e., at least one recursive call of FBT can be made in lines 8–10 with a non-empty vertex set. Initially the algorithm is called with $J = V_0$, where V_0 is the set of vertices in the initial graph. The non-emptyness condition can be trivially fulfilled by setting for each recursive call $J = V$, where V is the set of vertices of graph G on which the FBT algorithm is applied, i.e., the input graph. Later, in the context of the MEC algorithm (Section 2.4), we present an alternative choice for J, which in fact is the motivation for us introducing a candidate set to FBT in the first place.

As previously mentioned, the FBT algorithm involves a trimming step [26] (see Algorithm 1, line 1). This step eliminates the trivial SCCs consisting of a single vertex. The trimming procedure exploits topological sort elimination by starting in a vertex with zero in- or out-degree. As such vertex cannot be a part of a non-trivial SCC, they can be safely removed to avoid using them as pivots in the FBT search. Since the removal can create other trimming candidates, the procedure is iterated (in the method TRIM(V) in Algorithm 1) until there are no vertices for trimming left. Trimming is also used in our parallel SCC algorithm. Several studies [6, 22, 25] have shown that parallel SCC decomposition algorithms including Coloring heads off [28] and Recursive OBF [4], show inferior performance compared to the FBT algorithm.

Bloem *et al.* [7] presented an optimisation of Forward-Backward BFS as presented by Xie & Beerel. The optimisation takes into account that whenever either the forward or backward BFS has finished, but the other has not, then the latter can be restricted to those states that have been visited by the former. A version of FBT with this optimisation is described in Algorithm 2, and we refer to it as *Bounding* FBT (BFBT). In this version of the algorithm, we refer with *Ffront*, *Bfront* to the search frontier of the forward and backward BFS, respectively. Functions FWDBFSITERATION and BWDBFSITERATION perform one iteration of the forward and backward BFS, respectively, by moving the respective frontier to neighbouring states. Whenever one search has finished (condition of while loop at lines 6-8), the set *Converged* will contain all the states that were reached in the search that finished (lines 9-12). Next, either lines 13-14 or lines 15-16 will be executed, depending on which search still has work to do. Note that these searches are bounded to the area that has been searched by the search that terminated (conditions at lines 13 and 15).

BFBT has a clear advantage when the input graph has a structure where an FBT search would typically finish either one of the searches much sooner than the other. On the other hand, lines 19-20 show that there is less potential for increasing

Algorithm 2 Bounding FB with Trimming (BFBT)

```
Require: graph $G = (V, E)$, set $J \subseteq V$
Ensure: SCC decomposition of $G$ is given
    $V' \leftarrow$ Trim($V$) produces trivial SCCs
 2: if $V' \neq \emptyset$ then
    $pivot \leftarrow$ selectPivot($V' \cap J$)
 4:   $Ffront \leftarrow \{pivot\}$
      $Bfront \leftarrow \{pivot\}$
 6:   while $Ffront \neq \emptyset \wedge Bfront \neq \emptyset$ do
        $Ffront \leftarrow$ fwdBfsIteration($Ffront, F, (V', E)$)
 8:     $Bfront \leftarrow$ bwdBfsIteration($Bfront, B, (V', E)$)
      if $Ffront = \emptyset$ then
10:     $Converged \leftarrow F$
      else
12:     $Converged \leftarrow B$
      while $Ffront \cap B \neq \emptyset$ do
14:     $Ffront \leftarrow$ fwdBfsIteration($Ffront, F, (V', E)$)
      while $Bfront \cap F \neq \emptyset$ do
16:     $Bfront \leftarrow$ bwdBfsIteration($Bfront, B, (V', E)$)
      remove SCC $F \cap B$ from $V'$
18:   do in parallel
        BFBT($((V' \setminus Converged), E), J$)
20:     BFBT($((Converged \setminus (B \cap F)), E), J$)
```

the number of independent searches; in BFBT, due to the fact that one of the searches did not run its course, we can only launch up to two new BFBT instances in newly discovered regions, as opposed to three in FBT (Algorithm 1, lines 8-10). In Section 7, we report on experimental results using both FBT and BFBT, which makes it possible to draw some conclusions regarding their performance.

2.3 Sequential MEC Decomposition Algorithms

The basic sequential algorithm for MEC decomposition of MDP graph $G = (V, E)$ is based on an iterative SCC decomposition of G followed by transforming the SCCs into MECs [3, 16, 2]. The algorithm consists of the following stages:

1. Compute the SCC decomposition of G.
2. For each SCC C:
   (a) compute $U = \{v \in C \mid \forall \mu.\, E_\mu(v) \not\subseteq C\}$.
   (b) If $U \neq \emptyset$, remove $\mathrm{Attr}(U) \cap C$ from G. (cf. Lemma 1.)
   (c) Else, C is a MEC [2] (cf. Lemma 2). As justified by Lemma 1.2, remove $\mathrm{Attr}(C)$ for every C for which we established that C is a MEC.
3. Recursively compute the MEC-decompositions of the sub-MDP graphs obtained after the removal of the vertices in steps 2 and 3. (This is needed since the removal of the vertices might have destroyed the strong connectivity of some of the components.)

The first step of the algorithm, i.e., the SCC decomposition of the MDP graph, can be done in $O(m)$ time, where $m = |E|$ is the number of edges, e.g., using, e.g., Tarjan's algorithm [30]. The second step can be done in $O(m)$ time. There are

[2] Since G has at least one bottom SCC, i.e. at least one SCC satisfies this criterion.

at most $n = |V|$ iterations implied by step 3, since in each iteration at least one vertex is removed. This yields an overall time complexity of $O(m \cdot n)$. Recent works [13] and [15] present an adapted MEC-decomposition algorithm with time complexity $O(m \cdot \min(\sqrt{m}, n^{2/3}))$ and $O(n \cdot k^2.38 \cdot 2^k)$, respectively, where k is the so-called tree width of G. We base our GPU algorithm on the basic algorithm, since the recent algorithms involve steps that seem very hard to perform within the many-core paradigm of GPUs, like the lock-step search phase of [13].

2.4 Towards a Parallel Algorithm for MEC Decomposition

In the parallel version of the MEC algorithm we would like for SCC decomposition to use (a version of) Algorithm 1. Previously, it was assumed that $J = V$. However, on a GPU, it is particularly hard to let all the threads together make some random decision. This is explained in detail in Section 5.3. In fact, the smaller the number of threads that need to make such a decision, the better. Since on the GPU, threads will be mapped to states on a one-to-one basis, this means that it could be advantageous to select an alternative candidate set $J \subseteq V$ which is still non-empty, but as small as possible.

To achieve this, we use a modification of the (basic) MEC algorithm which is inspired by the sequential algorithm for MEC decomposition from [12]. Consider a recursive call of the MEC decomposition algorithm applied to graph $G = (V, E)$. Let $G' = (V', E')$ be a *region*, i.e., a subgraph of G obtained after the removal of the SCC attractors in Step 2 of the MEC algorithm and to which we are about to apply a recursive MEC (SCC) decomposition in Step 4. Let L denote the set of nodes that have been removed in the last (most recent) iteration of the algorithm in Step 2 of the MEC algorithm, after the SCC decomposition in Step 1. Let J be the "join" set of all nodes u such that $E(u) \cap L \neq \emptyset$.

For the correctness of the algorithm, we observe that the set J has to remain non-empty until all MECs of the input graph (V, E) are found. If in Step 2, $L = \emptyset$, then all found SCCs are MECs (since set U of vertices with outgoing transitions which are not e-closed is empty) and we are done. If $L \neq \emptyset$ then, since each vertex $v \in L$ is originally part of a (non-trivial) SCC C, there must be an edge (u, v) with $u \in C \setminus L$ and consequently $u \in J$.

However, set J does not have to be defined in such a way that it is guaranteed that each SCC decomposition procedure is complete; not all SCCs need to be identified by the decomposition procedure, and in fact, this is not guaranteed by J as it is defined above.

The modified version of the MEC algorithm that exploits the new definition of J is as follows, with initially $J = V$:

1′. Compute FBT(V, E, J).
2′. For each SCC C:
   (a) compute $U = \{v \in C \mid \forall \mu.\, E_\mu(v) \not\subseteq C\}$.
   (b) If $U \neq \emptyset$, remove Attr$(U) \cap C$ from G. (cf. Lemma 1.)
   (c) Else, C is a MEC [3] (cf. Lemma 2). As justified by Lemma 1.2, remove Attr(C) for every C for which we established that C is a MEC.

[3] Since G has at least one bottom SCC, i.e. at least one SCC satisfies this criterion.

$3'$. Compute sets L of the nodes that were removed in Step 2 and assign to J the set of the vertices that have an edge to a node in L. Recursively compute the MEC-decompositions of the sub-MDP graphs obtained after the removal of the vertices in steps 2 and 3. (This is needed since the removal of the vertices might have destroyed the strong connectivity of some of the components.)

The main motivation behind using a smaller subset of V (V') for J as defined above is to possibly accelerate the pivot finding in the SCC decomposition step. We revisit this issue later in Section 5.3. Finally, it has to be noted that since J as defined above does not guarantee that the SCC decomposition steps produce complete results, using J may lead to fewer search regions being discovered in search iterations, and therefore to less potential for parallelism. We report on our findings regarding this in Section 7.

3 GPU Basics

Harnessing the power of GPUs is facilitated by specific Application Programming Interfaces. In this article, we assume a concrete NVIDIA GPU architecture and the Compute Unified Device Architecture (CUDA) interface. Nevertheless, the algorithms that we present here can be straightforwardly applied to any architecture which provides massive hardware multithreading, supports the SIMT (Single Instruction Multiple Threads) model, and relies on coalesced access to the memory.

CUDA is an interface by NVIDIA which is used to program GPUs. CUDA extends C and FORTRAN. We use the C extension. GPU-specific features of CUDA include special declarations to explicitly place variables in the various types of memory (see Figure 2), predefined keywords containing the IDs of individual threads and blocks of threads, synchronization statements for cooperation between threads, run time API for memory management (allocation, deallocation), and statements to launch functions, referred to as *kernels*, on a GPU. In this section we give a brief overview of CUDA, adequate for presenting our results in subsequent sections. More details can be found in, for instance, [10, 35].

CUDA Programming Model. A CUDA program consists of a *host* program which runs on the Central Processing Unit (CPU) and (a collection of) CUDA kernels. Kernels, which describe the parallel parts of the program, are executed many times in parallel by different threads on the GPU device, and are launched from the host. Most GPUs have the restriction that at most one kernel can be launched at a time, but there are also GPUs available that allow to run multiple different kernels on different threads. When launching a kernel, the number of threads that should execute it needs to be specified. All those threads execute the same kernel, i.e. code. Since CUDA 5.0, dynamic parallelism is supported, meaning that besides the host launching kernels, also GPU threads can do this; in all cases, though, the number of threads that need to execute the kernel needs to be specified.

A thread is executed by a streaming processor (SP), see Figure 2. In general, GPU threads are grouped in blocks of a predefined size, usually a power of two. We refer to this size with *BlockSize*. A block of threads is assigned to a multiprocessor. Each thread block is uniquely identifiable by its block ID (referred to with the keyword *BlockId*) and analogously each thread is uniquely identifiable by its thread

Table 1 Explanation of CUDA terminology.

Term	**Explanation**
warp	a group of usually 32 threads running together in lock-step
block	a larger group of threads running on an SM
NrOfThreads	the total number of threads running on the GPU
BlockSize	size of a block of threads
BlockId	the unique ID of a block
ThreadId	the unique ID of a thread w.r.t. its block (between 0 and $BlockSize - 1$)
Global-ThreadId	the globally unique ID of a thread (between 0 and $NrOfThreads - 1$)
Lane	the unique ID of a thread w.r.t. its warp (between 0 and 31)

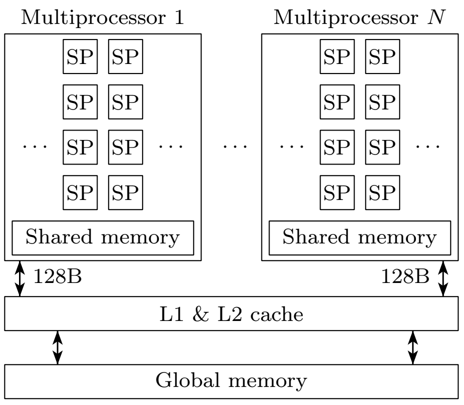

Fig. 2 Hardware model of CUDA GPUs.

ID (keyword *ThreadId*) within its block. Using these, it is possible to define other IDs, such as the GPU-global thread ID $\mathit{Global\text{-}ThreadId} = (\mathit{BlockId} \cdot \mathit{BlockSize}) + \mathit{ThreadId}$. The total number of threads running is defined by *NrOfThreads*. For clarity, an explanation of these and additional terms is given in Table 1.

CUDA Memory Model. Threads have access to different kinds of memory. Each thread has its own on-chip registers, access to which is very fast. Moreover, threads within a block can communicate via the *shared memory* of a multiprocessor, which is on-chip and also very fast. If multiple blocks are executed in parallel then the shared memory is equally split between them. All blocks have access to the *global memory* which is relatively large (usually up to 5 GB), but slow, since it is off-chip. Two caches called L1 and L2 are used to cache data read from the global memory. The host has read and write access to the global memory. Thus, the global memory is used for communication between the host and the kernel.

GPU Architecture. As already mentioned, the architecture of a GPU features a set of streaming multiprocessors (SMs). Each of those contains a set of SPs. The NVIDIA Kepler K20m, which we used for our experiments, has 13 SMs, each consisting of 192 SPs, which gives in total 2496 SPs. Furthermore, it has 5 GB global memory.

CUDA Execution Model. Threads are executed using the SIMT model. This means that each thread is executed independently with its own instruction address and

offsets	0	3	8	9	10	12	16	19	21

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
trans	s_1	s_2	s_4	s_1	s_2	s_2	s_3	s_3	s_2	s_3	s_5	s_6	s_2	s_4	s_6	s_7	s_4	s_5	s_6	s_2	s_3

Fig. 3 Example Compressed Sparse Row format graph storage. The transitions in *trans* are determined by their destination states. The arrays encode the graph induced by the MDP in Figure 1 without the actions and probabilities. One can see, for instance, that state s_0 is a source of transitions to s_1, s_2, and s_4, and from s_1 there is one transition to s_1, and there are two transitions to s_2 and two transitions to s_3.

local state (registers and local memory), but their execution is organized in groups of 32 called *warps* (see Table 1). The threads in a warp execute instructions in a synchronous manner, meaning that they move through the code in lock-step. This limits the possibilities for data races, but it also means that so-called *divergence* of thread executions can negatively impact performance of the computation. Consider the if-then-else construct **if C then A else B**. If the threads in a warp start executing this, and there are both threads for which **C** holds and threads for which it does not, then all the threads will together step through both alternatives **A** and **B**. The ones that do not need to execute **A** (or **B**) will have to 'go along' due to the SIMT model, but they will not actually execute it. Avoiding thread divergence is one of the main worries when implementing a program for the GPU.

Similarly, memory accesses of the threads in a single warp are serialized when they need to access separate parts of the global memory. If these accesses can be grouped together physically, i.e. if the accesses are coalesced, then the data can be obtained using a single fetch, thereby greatly improving the runtime. Hence, global memory access should be coalesced as much as possible. This is orthogonal to the fact that in graph decomposition algorithms, accessing transitions is irregular. Thus, achieving coalesced access is non-trivial. For sparse graphs, we propose a technique to reduce irregular memory access later in this section.

4 Related GPU Implementations

Sparse graphs are usually stored in a format based on the *Compressed Sparse Row (CSR)* format. An integer array *trans* of size $|E|$ is used to store all the transitions, in order of the source state IDs, and an array *offsets* consisting of $|V| + 1$ integers provides the start and end indices of the outgoing transitions of each source state, e.g. for state i, its outgoing transitions are stored in *trans* from position $\mathit{offsets}[i]$ up to and including $\mathit{offsets}[i+1]-1$. This encoding differs from CSR only in the fact that the CSR format is originally used to store sparse matrices, in which the row indices of the non-zero elements are also required, and are stored in a third array. In our case, this is not needed, but for clarity, we still refer to the format as CSR. The CSR representation of the graph induced by the MDP in Figure 1, without the actions and probabilities, is given in Figure 3.

The usual approach to perform a BFS-like search through a CSR description on a GPU involves the threads repeatedly scanning the *offsets* array using their ID, as in [21]. Such a GPU based algorithm is given in Algorithm 3. Note that GPU specific notions such as *NrOfThreads* have been defined in Section 3. Two

Algorithm 3 GPU Breadth-First Search iteration

```
Require: initial state is in search frontier
Ensure: if state $i$ is in search frontier, then the successors of $i$ are added to search frontier,
    and $i$ is moved to the explored set
    $stepsize \leftarrow 1$
 2: for ($i \leftarrow Global\text{-}ThreadId; i < |V|; i \leftarrow i + NrOfThreads$) do
        $srcinfo \leftarrow offsets[i]$
 4:     if inFrontier($srcinfo$) then
            $offsets[i] \leftarrow$ movetoExplored($srcinfo$)
 6:         $offset1 \leftarrow$ getOffset($srcinfo$)
            $offset2 \leftarrow$ getOffset($offsets[i + stepsize - (i \bmod stepsize)]$)
 8:         for ($j \leftarrow offset1$; $j < offset2$; $j \leftarrow j + stepsize$) do
                $t \leftarrow trans[j]$
10:             if $t \neq$ empty then
                    $tgtstate \leftarrow$ getTgtstate($t$)
12:                 $tgtinfo \leftarrow offsets[tgtstate]$
                    if isNew($tgtinfo$) then
14:                     $offsets[tgtstate] \leftarrow$ addToFrontier($tgtinfo$)
```

of the three highest bits in the *offsets* entries indicate whether the corresponding state is 1) in the search frontier or not and 2) has been explored or not.

Variable *stepsize* in line 1 of Algorithm 3 is related to input restructuring, i.e., an optimized version of the graph representation via *trans*, discussed in Section 5.1. Throughout this version of Algorithm 3 we assume $stepsize = 1$. In order to check the status of state i, the source state of the transitions we are going to explore, we copy $offsets[i]$ to *srcinfo* (line 3). We check if state i belongs to the frontier (line 4) by inspecting the highest bit of the variable *srcinfo*. If state i is in the frontier it is marked as explored by resetting the highest bit and setting the second highest bit of $offsets[i]$. After that all transitions of state i are explored. To this end first the offset interval corresponding to the transitions of i is established in lines 6 and 7. After that all transitions are inspected to possibly generate new frontier states (lines 8-14). It is checked at line 10 whether transition t has the special value **empty**. This is related to the optimisation described in Section 5.1, and can be ignored for now. The target state *tgtstate* of the inspected transition is extracted from t in line 11 and a copy of the *offsets* entry for *tgtstate* is saved in *tgtinfo*. In line 13 it is checked if the target state is new, i.e., it has not been visited yet. If this is the case, it is added to the frontier by setting the highest bit of the corresponding *offsets* entry. Note that any possible occurring data races due to multiple threads reaching the same successor state simultaneously can be considered benign; every thread executing line 14 tries to update $offsets[tgtstate]$ in the same way, namely by setting the highest bit.

Such an approach to BFS requires many complete scans of *offsets* to detect the current frontier and explore states. Since global memory is slow, this is a major performance bottleneck.

Li *et al.* [25] remark that a GPU BFS which avoids a one-to-one mapping between threads and nodes is preferable over the standard quadratic approach. In other words, approaches like the one of Merrill *et al.* [27], which uses a work queue, would be preferable. An important reason is that many threads otherwise idle, and with large differences in the out-degree of nodes, work imbalance tends to occur. With sparse matrices such as those underlying MDPs, however, this is not a big concern. The out-degree of most states tends to be similar, and small. In fact, in [29], an implementation of Merrill's approach does not result in further speedups for model checking problems, but it does require more memory. Therefore, we opt for the standard approach to do BFS on a GPU.

Pivot selection is an important step in SCC decomposition, which is non-trivial to implement efficiently on a GPU, since all threads need to agree on the pivots used for the newly discovered regions before launching new BFSs, and the regions need to be distinguishable by means of unique IDs. Several elaborate schemes for this have been presented. In [6], an additional array of size $|V|$ is used, and all threads assigned to states in regions that need to be searched try to write their ID to a common entry in this array. Determining which entry should be targeted is done using a region counting scheme and renumbering heuristics. Also in [25], such an array is used, but instead of racing to entries, a random number generator is implemented, state IDs are written to designated entries, and a prefix sum is used to count the number of new regions. Finally, Hong et al. [22] maintain set representations while doing the forward and backward BFSs, and use these to select pivots. We claim that our solution, which we explain in this section, is more elegant than earlier attempts, and at least as efficient. Instead of essentially trying to use a region counter, we simply use the pivot IDs themselves to identify regions, and our procedure requires no additional memory, instead using the *results* and *trans* arrays.

In addition to our new pivot selection, we also contribute compared to earlier work by using SM local caching of states, and restructuring the input to increase the number of coalesced memory accesses. Finally, we merge the frontier and explored set representations with the graph representation, thereby being more economic with the memory, and avoiding additional memory lookups.

5 SCC Decomposition on the GPU

5.1 Data Representation

For the encoding of a transition, first of all note that for our problems, the probability distributions in MDP graphs are not relevant, only 1) the target states, and 2) the distribution group a transition belongs to. In our implementations, we desire to work with 32-bit integers, as opposed to 64-bit integers, since it allows more efficient use of the global memory on our GPU. Hence, we assume that for each transition, an encoding of the group and the target state together fits in a 32-bit integer. Our program actually checks this: first, it is determined for the input what the maximum number of groups per source state is, say m. Then, the $log(m)$ highest bits of each transition integer are reserved for the group encoding.

To produce the desired output, i.e. the SCC decomposition, we allocate memory for another integer array *results* of size $|V|$. After decomposition, its content indicates which states belong to which SCC. Any two states i, j belong to the same SCC iff $results[i] = results[j]$.

Besides the original input, when memory allows, we also store the transposed MDP graph on the GPU. Since the original representation is tailored for a (forward) BFS, the transposed graph will be for a backward BFS. If there is not enough memory, then a kernel is available for scanning *offsets* and *trans* to perform a backward BFS, which is possible, but requires more memory accesses.

Finally, for bookkeeping purposes, we reserve the three highest bits in each entry of *offsets* and *results*. One bit of each entry i is used to indicate that state i is no longer involved in the current search iteration, i.e. it is already identified as part of a component. The two remaining bits in *offsets* and *results* entries are used to keep track of the search frontier and the set of explored states in the forward and the backward BFSs, respectively. We reason that this is acceptable: with this restriction, it is still possible to refer to 2^{29} states, i.e. about 537 million states. For a graph to be decomposed by our GPU implementation, $2 \cdot |V| + |E| + 1$ integers are needed if the transposed graph is not stored, and $3 \cdot |V| + 2 \cdot |E| + 2$ if it is. The vast majority of currently available GPUs have at most 5 GB global memory, which allows up to 1.3 billion integers to be stored, hence 29 bits is sufficient on those GPUs to refer to all the states of a graph that can be handled. In the future, when larger amounts of global memory is readily available (for instance, the NVIDIA Kepler K40 already has 12 GB memory), one can still switch from 32-bit to 64-bit integers to use our implementation for larger input graphs.

Restructuring input for coalesced memory access. In a BFS iteration, offsets are read in a coalesced way by the fact that the threads in a warp, with consecutive IDs, access an uninterrupted range of offsets. For the transitions in *trans*, though, this is a different matter, which is illustrated on the left in Figure 4. For the sake of clarity, we assume in this example that the warp size is 3, and we consider three states s_0 to s_2, and their outgoing transitions. In the figure, transition t_{00} is the first outgoing transition of state s_0, t_{10} is the first one of state s_1, and so on. Since the transitions are stored in separate blocks in *trans*, it is clear that access to *trans* will not be coalesced.

To fix this, we interleave the transition entries such that for all the states assigned to a warp, their first transitions are stored in an uninterrupted block, followed by all the second transitions, and so on. This allows to fetch transitions in a coalesced way. The drawback of this is that padding might be required to ensure that each thread accesses the same number of entries. On the right of Figure 4, the interleaved version of the example is given. We call a block of transitions ordered in this fashion which is assigned to a warp a *segment*. The amount of padding required is calculated per segment, so the locally maximum number of outgoing transitions determines the padding. The end of a segment is indicated in *offsets* at the first position of the next segment.

To avoid extensive padding, though, we use a hybrid representation. For a user-defined out-degree upper-bound u, which we call the segment *interval*, all the states with at most u outgoing transitions are renumbered to appear in the first part of *offsets* and *trans*, and all the other states are placed at the tail end. In the corresponding first part of *trans*, restructuring is applied, but on the tail part it is not. This allows to avoid that states with unusually many transitions cause the introduction of too many padding entries in the segment they are assigned to. For example, a hybrid representation with $u = 2$ of the example in Figure 4 is given in Figure 5. In that case, the amount of padding is reduced from three entries, as on the right in Figure 4, to only one entry. The drawback, though, is that accessing the transitions in the hybrid representation tends to require more separate data accesses. In the example, the transitions in the fully interleaved list can be read in three accesses, while the hybrid list requires five accesses.

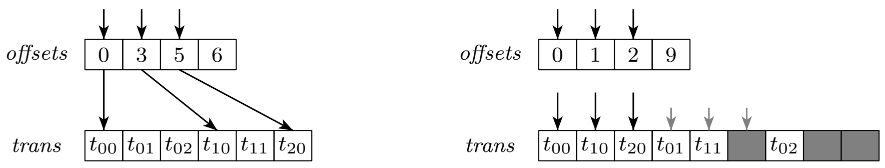

Fig. 4 Fetching transitions before and after restructuring.

In Algorithm 3, restructured transitions are supported when setting *stepsize* to the warp size. A hybrid representation using the segment interval would involve checking whether the transitions of a state are stored in a segment or not before exploring them, and setting *stepsize* appropriately.

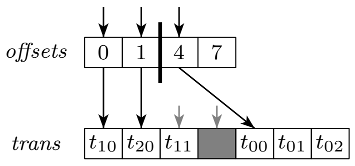

Fig. 5 Hybrid structure with $u = 2$, combining interleaved and non-interleaved storage.

5.2 GPU Search Mechanism

To illustrate our implementation of FBT for GPUs, we will discuss some of its more interesting aspects. Essentially, every step of Algorithm 1 is parallelised by means of a separate kernel. In addition to this, we also have a kernel for the combination of lines 4 and 5, i.e. the BFSs. In this hybrid kernel, iterations of both BFSs are performed simultaneously during a single scan of the offsets.

Algorithm 4 describes the GPU forward BFS. A local cache is allocated in shared memory. The size of this cache is defined in the host code, i.e. externally, as its declaration mentions. Its contents is initialised as empty. At lines 3-7, the offsets entries assigned to the executing thread are read and checked.

The approach to BFS as given in Algorithm 3 requires many complete scans of *offsets* to detect the current frontier and explore states. Since global memory is slow, this is a major performance bottleneck. To mitigate this, we have opted for using SM local state caches residing in the shared memory. The GPU-FWDBFS kernel accepts a given number of iterations *NrIters*. In the first iteration (lines 3-7), the usual scanning is performed, but in addition to being added to the frontier in the global memory, newly discovered states are added to the cache. After the first iteration, lines 8-13 are executed, in which the cache is scanned for exploration work.

In Algorithm 5, the EXPLORE procedure is described, which is in the implementation actually directly integrated with GPU-FWDBFS. First, *stepsize* is defined depending on whether the transitions belonging to state i to be explored reside in a segment or not. At line 4, *srcregion* stores the FBT search region (see Section 2.2) to which state i belongs. Starting at line 7, the successors of i are read. At line 12,

Algorithm 4 GPU-FWDBFS with local caching

```
Require: number of iterations NrIters
Ensure: NrIters local BFS iterations from the given search frontier have been performed
    extern volatile _shared_ unsigned int cache []
    <initialise cache>
    for (i ← Global-ThreadId; i < |V|; i ← i + NrOfThreads) do
        srcinfo ← offsets[i]
        if inFrontier(srcinfo) then
            offsets[i] ← movetoExplored(srcinfo)
            explore(srcinfo)
    for (iter ← 1; iter < NrIters; iter ++) do
        for j ← ThreadId; j < cachesize; j ← j + BlockSize do
            i ← cache[j]
            if i ≠ empty then
                cache[i] ← empty
                srcinfo ← offsets[i]
                if inFrontier(srcinfo) then
                    offsets[i] ← movetoExplored(srcinfo)
                    explore(srcinfo)
```

isActive checks if the search region of the target state of transition t has already been identified as an SCC in a previous round. This is the case if both the second and third highest bits of *results*[j] are set. If it is not part of a detected SCC, and both the source and target state of t are part of the same search region (line 14), where *tgtregion* represents the search region of the target state (line 13), then the target state is eligible for addition to the frontier. If its *offsets* entry indicates that the state is newly discovered, then, depending on the current search iteration, the target state is or is not added to the local cache (lines 16-22). Besides this, *nextIter* is set, which is read by the host after each search iteration to determine whether another iteration is required. Also, the target state is added to the search frontier (line 22). Finally, in the final iteration, no states are added to the cache, since after the final iteration, kernel execution will stop anyway, and the contents of the shared memory does not survive once a kernel has terminated.

Similar to GPU-FWDBFS, we also have a backward BFS variant operating on the transposed graph, if present, and a backward BFS variant operating on the original graph, which works different from Algorithm 5, since it involves in each iteration checking that from a state, the current frontier can be reached. Keeping track of the contents of the frontier and the set of explored states is done by using the bookkeeping bits in *results*. Besides this, we have a hybrid approach, in which both an iteration of the forward BFS and the backward BFS is performed. All these different versions allow to manage at the host level which searches should be performed in the next iteration, based on the feedback given by the threads.

BFBT. Inspired by the optimisation described in BFBT (Algorithm 2) regarding the bounding of one BFS once the other has terminated, we implemented a similar optimisation, which differs from the original one in that it refers to the global state of the SCC decomposition as a whole, as opposed to individual BFBT searches. This global version can be implemented straightforwardly by changing the condition for exploration of individual states in the appropriate procedures. For instance, in the forward BFS, we add a condition that a state may only be explored if it has already been explored in the backward BFS. This condition would be added to lines 5 and 14 in Algorithm 4. In the backward BFS code, we add a similar condition regarding the exploration history of the forward BFS. The hybrid search is left unchanged. These procedures can then be used as follows: we repeatedly apply the hybrid procedure on all the states in the graph, until there is no state from which we can continue either the forward or backward BFS. Depending on which BFS was terminated, we then repeatedly launch the procedure continuing the other BFS, which now is bound to the area that was explored by the BFS that terminated. Since the procedures are applied globally on all states, and therefore may involve multiple independent forward-backward searches being performed in parallel, the bounding of searches is only done once no search can continue the forward (or backward) BFS. This approach does not allow once search to be bounded, while another search running in parallel is not.

5.3 Pivot Selection

Finally, the other main challenge is in selecting pivots. After merging the results of the forward and backward BFS in the bookkeeping bits of *results*, we resolve this by hashing the current regions of states to locations in *trans*. In other words, we are effectively temporarily reusing the *trans* array as a hash table for pivot selection.

Algorithm 6 without the boxed code presents our pivot selection procedure. Note that state i belongs to search region $\mathit{results}[i]$, which is stored in *srcinfo* at line 2. At line 6, it is determined whether state i is eligible for becoming pivot. This is the case if i is both active, and not reached in both the forward and backward BFS of the previous FBT iteration. Here, reachedinBwd and reachedinFwd indicate whether the state has been reached in the backward or forward BFS, respectively. If i is eligible, a hash h is computed for it at line 17 (*leader* is currently not relevant, we return to this later). This hash is computed as $(\mathit{results}[i] + \textsc{reachedinBwd}(\mathit{results}[i]) + 2 \cdot \textsc{reachedinFwd}(\mathit{results}[i])) \bmod |E|$. Since this location may actually be beyond the bounds of *trans*, pivot selection is performed in several iterations, i.e., the procedure described in Algorithm 6 is invoked several times, each time with an incremented value *iter*. In each iteration, only the regions with a hash between $\mathit{iter} \cdot |E|$ and $(\mathit{iter} + 1) \cdot |E|$ are considered. Once a thread has determined the hash h, it will try to 'claim' the corresponding $\mathit{trans}[h]$ entry by atomically writing the ID of its state with the highest bit set to lock the entry (lines 24-25). The atomic compare-and-swap operation (atomicCAS) takes three arguments, namely the address where the new value must be written, the expected value at the address, and the new value. It only writes the new value if the expected value is encountered, and returns the encountered value, therefore a successful write has happened if t has been returned (line 26). Exactly one thread i will be able to do this, after which that thread will store the original $\mathit{trans}[h]$ entry temporarily in $\mathit{results}[i]$ (line 28). All other competing threads encounter a locked $\mathit{trans}[h]$ entry either at line 22 or line 26, and write this new pivot information into their *results* entries at line 37. The enforced data races using atomicCAS are used to pseudo-randomly choose pivots.

Finally, to revert *trans* back to its original content, after pivot selection, thread i needs to swap $\mathit{results}[i]$ and the unlocked $\mathit{trans}[h]$. This is not listed in Algorithm 6, since it can only be done after a global thread synchronisation, and therefore must be done in a different kernel. Note that with this approach, SCCs are actually identified by their pivots, and any number of pivots can be selected in parallel.

Optimisation 1 - Warp-aggregated Pivot Selection. Enforcing data races is an elegant way to pseudo-randomly select pivots, but requiring that many threads simultaneously perform atomic operations on the global memory can potentially result in a performance bottleneck. This is because atomic operations are scheduled to be performed in sequence, and hence reduce the level of parallelism.

To mitigate this effect, we use so-called *warp-aggregation* to ensure that within a warp, at most one thread per new search region will attempt to make its state a pivot. Warp-aggregation is achieved in general by the fact that communication among threads in a warp can be achieved instantaneously via their local registers. The boxed code in Algorithm 6 lists our extension to the pivot selection procedure to achieve this warp-aggregation. At lines 8-14, leaders are elected for each search region considered by threads in the warp. First, at line 8, the instruction $\_\mathsf{ballot}(1)$ produces a 32-bit bitmask indicating for each of the 32 threads in the warp whether or not they are active. Among these, the smallest warp ID or *Lane* (Table 1) j belonging to an active thread is selected using $\_\mathsf{ffs}()$. In fact, $\_\mathsf{ffs}()$ returns $j + 1$, 0 indicating that no thread is active. Hence, leader election continues until there are no more active threads (line 9).

At line 10, the $\_\mathsf{shfl}(v, i)$ instruction is used, which returns the value of variable v of thread i. So in this case, the new search region information is broadcasted from leader $j-1$ to all other threads in the warp. All threads considering the same search region choose $j - 1$ as their leader at lines 11-12, and subsequently break out of the while-loop, effectively disabling themselves until all threads in the warp have broken out. Finally, at line 14, the next leader is identified.

Once all leaders have participated in pivot selection (line 15), the results need to be propagated back to all other threads in the warp. This is done at lines 41-48 in a way similar to how the leaders are elected. Note that if the leader broadcasts an empty value (line 45 checks for this), then the leader was not able to participate in pivot selection yet, and must try again in the next pivot selection iteration (*iter* must be increased).

Optimisation 2 - Using set J. Another approach to try to optimise the pivot selection procedure is by reducing the number of states that are eligible for becoming pivot. Any such reduction has to ensure, however, that there are states eligible for becoming pivot as long as there is still work remaining, i.e., there are still SCCs or MECs to be discovered. In Section 2.2, we refer to this as the non-emptiness condition.

For MEC decomposition, we suggest to use the definition of J given in Section 2.4. There, it was argued that that definition satisfies the non-emptiness condition. The sets J can be used to reduce the number of pivot candidates in the SCC decomposition step of one MEC decomposition iteration based on the results of the previous MEC decomposition iteration. In other words, this means that this proposed optimisation, in contrast to the previous one, is not applicable for 1) stand-alone SCC decomposition, 2) the very first SCC decomposition step performed for MEC decomposition.

Algorithm 5 EXPLORE with local caching for GPU

```
Require: offset entry $srcinfo$ of a state $i$
Ensure: the successors of $i$ are added to the search frontier
  $stepsize \leftarrow 1$
  if $i < 32 \cdot \#segments$ then
    $stepsize \leftarrow 32$
  $srcregion \leftarrow$ GETREGION($results[i]$)
  $offset1 \leftarrow$ GETOFFSET($srcinfo$)
  $offset2 \leftarrow$ GETOFFSET($offsets[i + stepsize - (i \bmod stepsize)]$)
  for ($j \leftarrow offset1$; $j < offset2$; $j \leftarrow j + stepsize$) do
    $t \leftarrow trans[j]$
    if $t \neq$ empty then
      $tgtstate \leftarrow$ GETTGTSTATE($t$)
      $r \leftarrow results[j]$
      if ISACTIVE($r$) then
        $tgtregion \leftarrow$ GETREGION($r$)
        if $srcregion = tgtregion$ then
          $tgtinfo \leftarrow offsets[tgtstate]$
          if ISNEW($tgtinfo$) then
            if $iter < NrIters - 1$ then
              if $\neg$STOREINCACHE($tgtstate$) then
                $nextIter \leftarrow$ true
            else
              $nextIter \leftarrow$ true
            $offsets[tgtstate] \leftarrow$ ADDTOFRONTIER($tgtinfo$)
```

The optimisation can be added to Algorithm 6 by extending the condition checked at line 6 with the condition that state i must be in J. Membership in J can be maintained by using one of the bookkeeping bits for this purpose (in fact, in the implementation, there is one bit still available for this purpose in the *offsets* entries). What remains is to update the set J after every MEC iteration. Initially, all states are in J. Once the sets U and the attractor sets have been computed (see Section 6), one additional scan of the states needs to be performed to identify the states that have at least one transition leading to a state set for removal.

6 MEC Decomposition on the GPU

Our GPU implementation for MEC decomposition is based on the algorithm presented in Section 2.4. For step 1, we use our GPU SCC decomposition. For step 2, we first reset the second and third highest bookkeeping bits in *results* to reuse them as follows: one bit is used to indicate that a state should be removed, and the other bit is used to mark newly discovered MECs. First, a single scan of the input suffices to identify the sets U of the various SCCs. Pseudo-code for this is presented in Algorithm 7. Before this code is executed, for each search region with pivot j, entry $trans[j]$ is locked to indicate that no state has yet been found in that region belonging to U. Each state i active in the search but not in U (line 3) is inspected to determine whether in each outgoing transition group there is at least one transition going out of the SCC containing i. This is done at lines 12-23, using r and *found* to indicate whether a group has been found that does not contain a transition leaving the SCC, and whether such a transition has been found for the currently explored group, respectively. Finally, at lines 26-27, state i is added to

Algorithm 6 GPU Pivot selection (warp-aggregated with boxed code)

Require: iteration index *iter*

Ensure: in each search region with at least one pivot candidate, exactly one candidate is promoted to pivot, and all other candidates in the region know its identity.

```
for ($i \leftarrow Global\text{-}ThreadId; i < |V|; i \leftarrow i + NrOfThreads$) do
    $srcinfo \leftarrow results[i]$
    $srcregion \leftarrow$ getRegion($srcinfo$)
    $inF \leftarrow$ reachedinFwd($srcinfo$)
    $inB \leftarrow$ reachedinBwd($srcinfo$)
    if isActive($srcinfo$) $\wedge \neg(inF \wedge inB)$ then
        $leader \leftarrow i$
        $j \leftarrow$ _ffs(_ballot(1))
        while $j > 0$ do
            $tgtinfo \leftarrow$ _shfl($srcinfo, j - 1$)
            if $tgtinfo = srcinfo$ then
                $leader \leftarrow j - 1$
                break
            $j \leftarrow$ _ffs(_ballot(1))
        if $Lane = leader$ then
            $write \leftarrow$ false
            $h \leftarrow 3 \cdot srcregion + inB + 2 \cdot inF$
            $srcregion \leftarrow$ empty
            if $h/|E| = iter$ then
                $h \leftarrow h \bmod |E|$
                $t \leftarrow trans[h]$
                if $\neg$isLocked($t$) then
                    $srcregion \leftarrow i$
                    lock($srcregion$)
                    $t' \leftarrow$ atomicCAS(&($trans[h]$), $t$, $srcregion$)
                    if $t' = t$ then
                        lock($t'$)
                        $results[i] \leftarrow t'$
                        $contpost \leftarrow$ true
                    else
                        $srcregion \leftarrow t'$
                        $write \leftarrow$ true
                else
                    $srcregion \leftarrow t$
                    $write \leftarrow$ true
                if $write$ then
                    $results[i] \leftarrow srcregion$
            else
                $srcregion$
                $cont \leftarrow$ true
        $j \leftarrow$ _ffs(_ballot(1))
        while $j > 0$ do
            $tgtregion \leftarrow$ _shfl($srcregion, j - 1$)
            if $leader = j - 1$ then
                if $tgtregion \neq$ empty then
                    $results[i] \leftarrow tgtregion$
                break
            $j \leftarrow$ _ffs(_ballot(1))
```

Algorithm 7 GPU detection of sets U

```
Require: an SCC decomposition
Ensure: for each SCC $C$, $U = \{v \in C \mid \forall\mu.\, E_\mu(v) \nsubseteq C\}$ is determined. Each state $v \in U$ has
    been marked as a member of $U$.
    for ($i \leftarrow$ Global-ThreadId; $i < |V|$; $i \leftarrow i +$ NrOfThreads) do
        $srcregion \leftarrow results[i]$
        if isActive($srcregion$) $\wedge \neg$inU($srcregion$) then
            $offset1 \leftarrow$ getOffset($offsets[i]$)
            $offset2 \leftarrow$ getOffset($offsets[i + stepsize - (i \bmod stepsize)]$)
            $r \leftarrow$ false
            $g \leftarrow$ empty
            $found \leftarrow 2$
            for ($j \leftarrow offset1$; $j < offset2 \wedge \neg r$; $j \leftarrow j + stepsize$) do
                $t \leftarrow$ unlocked($trans[j]$)
                if $t \neq$ empty then
                    $tgtstate \leftarrow$ getState($t$)
                    $g' \leftarrow$ getGroup($t$)
                    $tgtregion \leftarrow results[tgtstate]$
                    if $\neg$isLocked($tgtregion$) then
                        if $g' \neq g$ then
                            if $found = 0$ then
                                $r \leftarrow$ true
                            else
                                $g \leftarrow g'$
                                $found \leftarrow 0$
                        if $\neg r \wedge$ getRegion($tgtregion$) $\neq srcregion$ then
                            $found = 1$
            if $\neg r$ then
                if $found = 1$ then
                    unlock($trans[srcregion]$)
                    $results[i] \leftarrow$ addtoU($srcregion$)
```

U if the aforementioned condition holds. Whenever a state i in the SCC with ID *srcregion* is identified to be in U, we unlock entry *trans*[*srcregion*] to indicate that there are states in the SCC that are also in U, and hence the SCC cannot be a MEC (line 26).

Having detected the U, we compute the intersections of the attractor sets of the U and the SCCs that they belong to; states in those sets are marked for removal. In step 3, *results* is scanned and all entries with region *srcregion* and *trans*[*srcregion*] unlocked are marked as being in a MEC. Subsequently, we repeatedly compute the attractor sets of those MECs and mark the entries for removal. Concluding, in a single scan, locked *trans* entries are unlocked, to be removed *results* entries are set to the **empty** value (and their *offsets* entry is locked), and discovered MECs are locked as well.

It is important to note that SCCs discovered in a MEC decomposition iteration must necessarily be subsets of SCCs discovered in the previous iteration. This means that we can reuse earlier results to select multiple pivots at the start of an iteration, thereby starting multiple FBT searches in parallel.

7 Experiments

We conducted experiments to measure the performance of our implementations using a representative set of benchmark models taken from the standard distri-

Table 2 Structural properties of the graphs. **|V|**: number of vertices, **|E|**: number of edges, **av. out**: average out-degree, **max. out**: maximum out-degree, **#SCCs**: number of SCCs. **Explanation of the model instances**: wlan.2500: wlan6 with TRANS-TIME-MAX=2,500, phil_lss.5.10: phil_lss with 5 phils, $K = 10$, coin8.3: coin8 with $K = 3$, mutual7.13: mutual with 7 processes, 13 states each, zeroconf_dl.F.200.1k.6: zeroconf_time_bounded with *reset* = **false**, $T = 200$, $N = 1,000$, $K = 6$, firewire_dl.800.36.(0.2): firewire impl/deadline with *deadline* = 800, *delay* = 36, *fast* = 0.2.

	Model	|V|	|E|	av. out	max. out	#SCCs
1	wlan.2500	12.6M	28.1M	2.23	129	12.5M
2	phil.7	11.0M	98.5M	8.97	14	1
3	diningcrypt.10	42.9M	279.4M	6.51	20	42.9M
4	test-and-set.7	51.4M	468.5M	9.12	17	4,672
5	leader.7	68.7M	280.5M	4.08	14	42.2M
6	phil_lss.5.10	72.9M	425.6M	5.84	10	1
7	coin.8.3	87.9M	583.0M	6.63	16	5.4M
8	mutual.7.13	76.2M	653.7M	8.58	14	1
9	zeroconf_dl.F.200.1k.6	118.6M	273.5M	2.31	10	118.6M
10	firewire_dl.800.36.(0.2)	129.3M	293.6M	2.27	5	129.3M

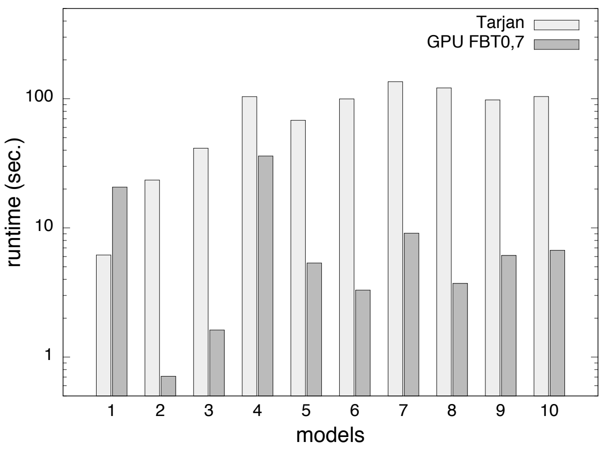

Fig. 6 Runtimes (in logscale) of SCC decomposition using a single-core implementation of Tarjan's algorithm and the overall best GPU configuration ('GPU FBT0,7')

bution of the PRISM model checker and additional models provided through its dedicated website.[4] In fact, we have selected all available MDP models that were scalable to interesting proportions while not requiring more memory than our GPU could handle, and were accepted by the latest version of PRISM. All experiments were performed on machines running CentOS Linux, with an Intel E5-2620 2.0 GHz CPU, 64 GB RAM, and an NVIDIA Kepler K20m GPU. This GPU has 2,496 cores and 5 GB global memory. It should be noted that given the range of features of CUDA that we use, the implementation requires that the NVIDIA GPU on which it is executed has at least computation capability 3.0.

[4] All relevant material is available at `http://www.win.tue.nl/~awijs/gpudecompose`.

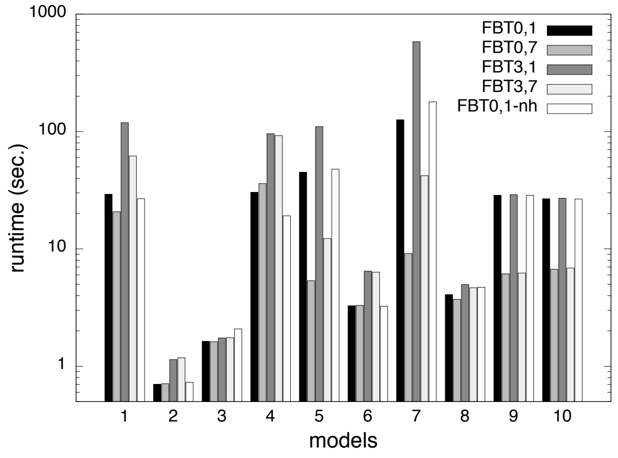

Fig. 7 Runtimes (in logscale) for SCC decomposition for various GPU configurations Fi, j, where i is the number of search iterations per BFS kernel launch, and j is the interval used for input restructuring. In FBT0, 1-nh, the hybrid forward-backward kernel has been disabled

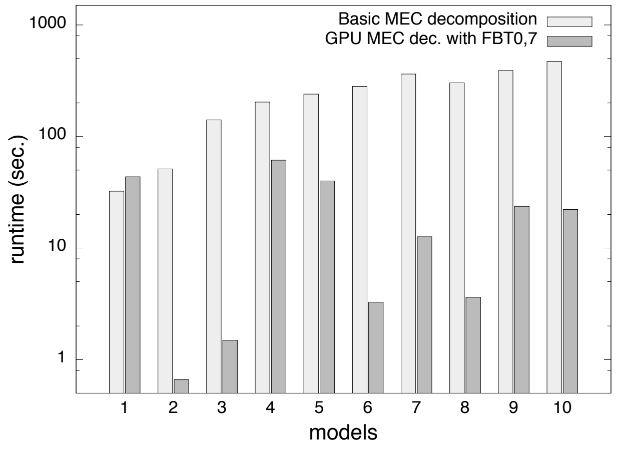

Fig. 8 Runtimes (in logscale) for MEC decomposition comparing the basic decomposition algorithm against the overall best GPU configuration

For all GPU experiments, we launched $|V|/512$ blocks of 512 threads each, i.e. one thread per state. This keeps the amount of work per thread minimal, and does not introduce idle threads that keep the scheduler busy.

Table 2 presents the graph characteristics of the cases. The 'av. out' column provides the average out-degree, the 'max. out' column the maximum out-degree, and the '#SCCs' column displays the number of SCCs in the graph. Most graphs have a very particular structure; several consist practically entirely of trivial SCCs, and others are a single SCC. We have not preselected any models, so it is interesting to note this phenomenon. It merits further study whether most MDP problems boil down to MDP graphs of one of these types.

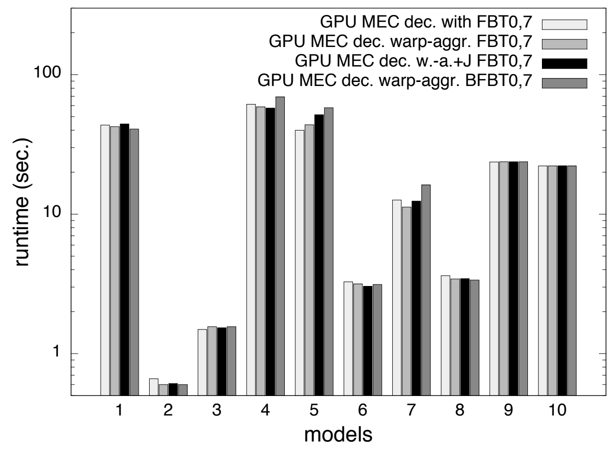

Fig. 9 Runtimes (in logscale) for MEC decomposition comparing different pivot selection procedures in FBT0,7, and a version of BFBT0,7 with warp-aggregation

Figure 6 presents our SCC decomposition runtime results comparing a single-core CPU implementation of Tarjan's SCC decomposition algorithm with the best configuration of our GPU FBT implementation. FBT0,7 represents GPU FBT with 0 search iterations per BFS kernel launch using the local cache, i.e., the cache is not used, and 7 being the interval (out-degree upperbound) used for restructuring the input.

For graphs consisting of only one SCC, speedups of around 30 times can be observed. This is not surprising, since these can be analysed in a single GPU search iteration. When there are many trivial SCCs present, the trimming procedure is very influential. The efficiency of the trimming procedure is bound by the average out-degree of a graph; the more connected a trivial SCC state is to other states, the more potential there is for detecting other trivial SCCs in the next trimming iteration. For this reason, the *diningcrypt* case can be decomposed quickly compared to the *zeroconf* and *firewire* cases.

In Figure 7, the results for various configurations of our GPU implementation are presented. Concerning *zeroconf*, *firewire*, and cases with many non-trivial SCCs, it can be observed that the input restructuring works very well (F0,7). In most cases, speedups of about 15 times can be observed. This is significant when considering that in related work [6], only speedups up to 5-6 times were measured for graphs representing model checking problems. Besides the restructuring, the new pivot selection procedure and the data representation likely also play a role in the improved speedup, but it is hard to determine how much, since these are core aspects of our implementation that we cannot easily disable. An experimental comparison with the work of [6] seems useful, however their implementation cannot handle graphs of similar size, due to the fact that eight bits are used per integer for bookkeeping, whereas we only use three. In addition, their implementation does not accept MDP graphs, so some reimplementation work would be required. It is clear, however, that coalesced data access, which is improved by using the restructuring option, is the main cause for the improved speedups.

We also performed some controlled experiments in which we disabled the hybrid search kernel (F0,1-*nh*). These show that using the hybrid kernel at best only causes a minor speedup. In some cases, disabling it even results in speedups, because it results for those particular graph structures in fewer memory accesses.

The contribution of the local caches is minimal (cases F3,1 and F3,7), and in most cases using them causes a slowdown. This is probably due to the fact that by using caches, states are assigned to thread blocks in a way that depends on the structure of the graph, as opposed to a direct one-to-one mapping. When threads visit unexplored states, they add them to the cache and thereby claim them for exploration. Which states get to be claimed by which blocks is therefore dependent on the graph structure. In turn, which *offsets* and *trans* entries will be accessed by which blocks in the next search iteration is unknown beforehand, and thus we can assume that these accesses will not be coalesced.

An overall negative result has been obtained for *wlan*. Its graph has a structure which considerably limits the trimming procedure. It both has a low average out-degree and only a few states from which trimming can be instantiated.

In Figure 8, results for MEC decomposition are presented. Speedups up to 79 times were measured. The cause for the increased speedups is that the additional steps after SCC decomposition can be performed extremely efficient in parallel on a GPU, since they require a single BFS-like analysis of the states and transitions, whereas such a BFS search on the CPU is much slower.

Finally, Figure 9 presents our runtime results when applying FBT0,7-based MEC decomposition with the two proposed pivot selection optimisations suggested in Section 5.3, and a MEC decomposition procedure using BFBT0,7 with warp-aggregation. As can be observed, the differences in runtime performance are rather small, but some meaningful conclusions can be drawn. First of all, warp-aggregation leads most of time to a speedup. Given the time spent in the overall computation on pivot selection, the acquired speedup is actually quite remarkable; most of the runtime is spent performing the BFS searches and the trimming, but as noted, the atomic operations performed in pivot selection do present a bottleneck. Leveraging this bottleneck has a noticeable effect on the runtimes. Of course, this does not hold for the cases where trimming suffices to detect practically all the SCCs, such as in the cases 9 and 10.

The optimisation using the J sets, however, has in a number of cases a negative effect, and can most of the time not provide any additional speedups when used together with warp-aggregation, which is the setup we investigated. This seems to suggest that the loss of potential parallelism, which we refer to in Section 2.4, in practice outweighs the potential benefit of having fewer states (and therefore threads) competing to become pivot.

Also regarding BFBT, in a number of cases, the loss of potential parallelism (due to the number of identified regions not growing as quickly in BFBT as in FBT) seems to outweigh the ability to more efficiently perform FBT searches. This is interesting, since it demonstrates that not all optimisations of a given sequential algorithm necessarily are also improvements for a parallel version of it.

8 Conclusions

We presented GPU algorithms for finding SCCs and MECs in sparse graphs that are based on FBT and a bounding version of it. The implementations exhibit speedups of 15-30 times for SCC decomposition and up to 79 times for MEC decomposition. A critical improvement for SCC decomposition compared to related work is achieved by improving (coalesced) data access. Other causes are a new pivot selection procedure and the chosen data representation, while techniques such as local caching do not lead to performance improvement. Furthermore, we have explained a number of potential optimisations to further improve pivot selection. Both defining smaller candidate sets and using BFBT instead of FBT did not result in consistently faster runtimes. On the other hand, warp-aggregation does lead to relatively good improvements of the runtimes, certainly considering the amount of time that pivot selection takes as part of the overall decomposition procedure.

The extra steps required for MEC decomposition are very suitable for parallelisation on GPUs, which explains the large speedups overall.

For future work, we plan to address similar problems in probabilistic model checking [3], and to integrate the algorithms in model checking tools. Furthermore, besides BFBT, a number of other optimisations have been proposed for FB-based algorithms to find SCCs, such as the one using so-called *spine-sets* described by Gentilini *et al.* [20]. It would be interesting to investigate whether any of those optimisations can effectively be used in a GPU implementation.

References

1. Ábrahám E, Jansen N, Wimmer R, Katoen JP, Becker B (2010) DTMC model checking by SCC reduction. In: QEST, IEEE Computer Society, pp 37–46
2. de Alfaro L (1998) How to specify and verify the long-run average behavior of probabilistic systems. In: LICS, IEEE Computer Society, pp 454–465
3. Baier C, Katoen JP (2008) Principles of Model Checking. The MIT Press
4. Barnat J, Moravec P (2007) Parallel Algorithms for Finding SCCs in Implicitly Given Graphs. In: FMICS/PDMC, Springer, LNCS, vol 4346, pp 316–330
5. Barnat J, Brim L, Ceska M, Lamr T (2009) CUDA Accelerated LTL Model Checking. In: ICPADS, IEEE, pp 34–41
6. Barnat J, Bauch P, Brim L, Ceska M (2011) Computing Strongly Connected Components in Parallel on CUDA. In: IPDPS, IEEE, pp 544–555
7. Bloem R, Gabow H, Somenzi F (2006) An Algorithm for Strongly Connected Component Analysis in n log n Symbolic Steps. Formal Methods in System Design 28(1):37–56
8. Bloem R, Gabow HN, Somenzi F (2006) An Algorithm for Strongly Connected Component Analysis in $n \log n$ Symbolic Steps. Formal Methods in System Design 28(1):37–56
9. Bošnački D, Edelkamp S, Sulewski D, Wijs A (2010) GPU-PRISM: An Extension of PRISM for General Purpose Graphics Processing Units. In: PDMC 2010, IEEE, pp 17–19
10. Bošnački D, Edelkamp S, Sulewski D, Wijs A (2011) Parallel Probabilistic Model Checking on General Purpose Graphic Processors. STTT 13(1):21–35

11. Bruyère V, Filiot E, Randour M, Raskin JF (2014) Meet your expectations with guarantees: Beyond worst-case synthesis in quantitative games. In: STACS, Schloss Dagstuhl, LIPIcs, vol 25, pp 199–213
12. Chatterjee K, Henzinger M (2011) Faster and Dynamic Algorithms for Maximal End-Component Decomposition and Related Graph Problems in Probabilistic Verification. In: SODA, SIAM, pp 1318–1336
13. Chatterjee K, Henzinger M (2012) An $O(n^2)$ Time Algorithm for Alternating Büchi Games. In: SODA, SIAM, pp 1386–1399
14. Chatterjee K, Henzinger M (2014) Efficient and dynamic algorithms for alternating Büchi games and maximal end-component decomposition. J ACM 61(3):15
15. Chatterjee K, Łącki J (2013) Faster Algorithms for Markov Decision Processes with Low Treewidth. In: CAV, Springer, LNCS, vol 8044, pp 543–558
16. Courcoubetis C, Yannakakis M (1995) The complexity of probabilistic verification. J ACM 42(4):857–907
17. Dijkstra EW, Feijen WHJ (1988) A Method of Programming. Addison-Wesley
18. Edelkamp S, Sulewski D (2010) Efficient Explicit-State Model Checking on General Purpose Graphics Processors. In: SPIN, Springer, LNCS, vol 6349, pp 106–123
19. Fleischer L, Hendrickson B, Pinar A (2000) On Identifying Strongly Connected Components in Parallel. In: IPDPS Workshop Irregular 2000, Springer, LNCS, vol 1800, pp 505–511, URL `http://dl.acm.org/citation.cfm?id=645612.663154`
20. Gentilini R, Piazza C, Policriti A (2003) Computing Strongly Connected Components in a Linear Number of Symbolic Steps. In: SODA, ACM/SIAM, pp 573–582
21. Harish P, Narayanan P (2007) Accelerating Large Graph Algorithms on the GPU Using CUDA. In: HiPC, Springer, LNCS, vol 4873, pp 197–208
22. Hong S, Rodia N, Olukotun K (2013) On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-World Graphs. In: SC'13, ACM, p 92
23. Kwiatkowska M, Norman G, Parker D (2011) PRISM 4.0: Verification of Probabilistic Real-time Systems. In: CAV, Springer, LNCS, vol 6806, pp 585–591
24. Kwiatkowska MZ, Parker D, Qu H, Ujma M (2014) On incremental quantitative verification for probabilistic systems. In: Voronkov A, Korovina MV (eds) HOWARD-60: A Festschrift on the Occasion of Howard Barringer's 60th Birthday, EasyChair, pp 245–257
25. Li G, Zhu Z, Cong Z, Yang F (2014) Efficient Decomposition of Strongly Connected Components on GPUs. Journal of Systems Architecture 60(1):1–10
26. McLendon III W, Hendrickson B, Plimpton S, Rauchwerger L (2005) Finding Strongly Connected Components in Distributed Graphs. J Parallel Distrib Comput 65(8):901–910
27. Merrill D, Garland M, Grimshaw A (2012) Scalable GPU Graph Traversal. In: PPoPP, ACM, pp 117–128
28. Orzan S (2004) On distributed verification and verified distribution. PhD thesis, Free University of Amsterdam
29. Stuhl M (2013) Computing Strongly Connected Components with CUDA. Master's thesis, Masaryk University

30. Tarjan R (1972) Depth-First Search and Linear Graph Algorithms. SIAM J Comput 1(2):146–160
31. Ummels M, Wojtczak D (2011) The Complexity of Nash Equilibria in Stochastic Multiplayer Games. Logical Methods in Computer Science 7(3)
32. Wang C, Bloem R, Hachtel GD, Ravi K, Somenzi F (2006) Compositional SCC analysis for language emptiness. Formal Methods in System Design 28(1):5–36
33. Wijs A (2015) GPU Accelerated Strong and Branching Bisimilarity Checking. In: TACAS'15, Springer, LNCS, vol 9035, pp 368–383
34. Wijs A, Bošnački D (2012) Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking. In: SPIN, Springer, LNCS, vol 7385, pp 98–116
35. Wijs A, Bošnački D (2014) GPUexplore: Many-Core On-The-Fly State Space Exploration. In: TACAS, Springer, LNCS, vol 8413, pp 233–247
36. Wijs A, Katoen JP, Bošnački D (2014) GPU-Based Graph Decomposition into Strongly Connected and Maximal End Components. In: CAV, Springer, LNCS, vol 8559, pp 309–325
37. Xie A, Beerel P (1999) Implicit Enumeration of Strongly Connected Components. In: CAD'99, pp 37–40