GPU-Based Graph Decomposition into Strongly Connected and Maximal End Components

Anton Wijs*[1], Joost-Pieter Katoen[2], and Dragan Bošnački[1]

[1] Eindhoven University of Technology, The Netherlands
[2] RWTH Aachen University, Germany

Abstract. This paper 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 counterparts in several case studies.

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 $\mathsf{EG}\,\varphi$ as well as for verifying fair CTL [1, Ch. 6] and checking language emptiness [2]. The high relevance of SCCs has led to various dedicated variants of Tarjan's classical algorithm [3] such as symbolic [4] and a plethora of parallel [5–7] algorithms. In the context of probabilistic model checking, a generalisation of SCCs – known as maximal end components (MECs) – play a pivotal role [8, 9]. 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

* 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.

to reside when playing against a probabilistic adversary. MEC decomposition of MDPs is typically a pre-processing step of probabilistic model checking to determining almost-sure limiting properties [1, Ch. 10]. Other applications include the analysis of multi-player stochastic games [10] as well as recent approaches to combined worst-case and expected value objectives for mean pay-off games [11]. Improvements of the traditional sequential algorithms for determining MECs [1, 8, 9] have been reported [12] and were tailored to MDPs with low tree-width [13].

In this paper, we provide new algorithms to efficiently decompose graphs into SCCs and MECs by exploiting GPUs (Graphical Processing Units). Our decomposition algorithms build upon three key principles. First, inspired by the Forward-Backward algorithm (FB) [14], each thread combines *a forward and a backward reachability search* so as to identify SCCs. Previous work on GPU-based SCC decomposition [5–7] 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. 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 [5] 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 [1, 8, 9]. To the best of our knowledge, this is the first GPU-based MEC decomposition. We implemented our algorithms using CUDA[3] for NVIDIA GPUs, and ran them on examples of the PRISM benchmark suite [15]. 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. [5]) 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 [16–18], 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 [16, 20, 19]. More recently, we presented an on-the-fly search algorithm for standard model checking running entirely on GPUs [21].

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

Organisation of the paper. Section 2 treats the basics of MDPs, MECs and relevant SCC and MEC decomposition algorithms. Section 3 gives a detailed account of our GPU algorithms focussing on our main design choices. Section 4 presents the experimental results, and Section 5 concludes.

2 Preliminaries

This section gives an introduction to the main concepts of MDPs and MECs [1, Ch. 10], presents the parallel FB algorithm for SCC decomposition [14] and the standard sequential algorithm for MEC decomposition [8, 9] 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 consider as a single-player game in which the system plays against a random adversary. An MDP naturally induces a digraph in the following sense.

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 some $\mu \in T(u)$.*

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 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 $G{\uparrow}C$, i.e., the graph $G{\uparrow}C = (C, (C \times \Delta(C) \times C) \cap E)$, 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 $G{\uparrow}U$ is strongly connected and every $u \in U$ is e-closed for U.*

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.

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 *$Attr(U)$ of $U \subseteq V$ is defined as $Attr(U) = \bigcup_{i \geq 0} U_i$ 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. The MEC-decomposition algorithm discussed later on exploits the following two results from [22]. The first result identifies 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 $U = \{v \in C \mid \forall \mu.\, E_\mu(v) \not\subseteq C\}$ 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 $X \neq C$ of G it holds that $Z \cap X = \emptyset$.*

The second result from [22] provides a sufficient criterion for an SCC to be a MEC.

Lemma 2 (Closed SCCs are MECs). *An SCC C of the MDP graph $G = (V, E)$ with $E(v) \subseteq C$ for all $v \in C$, is a MEC.*

A corollary of Lemma 2 is that every bottom SCC, i.e., an SCC C such that all transitions from C lead back to 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 [3] and Dijkstra [23] 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.* [14]. 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.

Algorithm 1 FB with Trimming (FBT)

```
Require: graph G = (V, E)
Ensure: SCC decomposition of G is given
    V' ← Trim(V) produces trivial SCCs
 2: if V' ≠ ∅ then
        pivot ← selectPivot(V')
 4:     F ← fwdBfs(pivot, (V', E))
        B ← bwdBfs(pivot, (V', E))
 6:     remove SCC F ∩ B from V'
        do in parallel
 8:         FBT(((F \ B), E))
            FBT(((B \ F), E))
10:         FBT(((V \ (B ∪ F)), E))
```

The Forward-Backward (FB) algorithm starts by (randomly) selecting a *pivot* vertex p (see Alg. 1, line 3). 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 (Alg. 1, lines 4-5). The intersection of the vertices reached via the forward and backward BFSs constitutes an SCC (and is removed, Alg. 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, FB 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. The FB algorithm can be improved by *trimming* [24] (see Alg. 1, line 1). This step eliminates the trivial SCCs consisting of only one 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 FB search. Since the removal can create other trimming candidates, the procedure is iterated (in the method $\textsc{Trim}(V)$ in Alg. 1) until there are no vertices for trimming left. Trimming is also used in our parallel SCC algorithm. Several studies [5–7] have shown that parallel SCC decomposition algorithms including Coloring heads off [25] and Recursive OBF [26], show inferior performance compared to the FBT algorithm.

2.3 Sequential MEC Decomposition Algorithms

The basic sequential algorithm for MEC decomposition of MDP graph $G = (V, E)$ is based on iterative SCC decomposition of G followed by transforming the SCCs into MECs [1, 8, 9]. The algorithm consists of the following stages:

1. Compute the SCC decomposition of G. For SCC C, let $U = \{v \in C \mid \forall \mu.\, E_\mu(v) \not\subseteq C\}$.

2. If $U \neq \emptyset$, remove $\mathrm{Attr}(U) \cap C$ from G. (cf. Lemma 1.)
3. Every SCC C without an outgoing edge is a MEC [4] (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.
4. 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 [3]. The second step can be done in $O(m)$ time. There are 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 [22] and [13] 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 [22].

3 GPU-Based Graph Decomposition Algorithm

3.1 GPU Basics

Harnessing the power of GPUs is facilitated by specific Application Programming Interfaces. In this paper, 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 1), 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, [16, 21].

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

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

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. Each thread is executed by a streaming processor (SP), see Figure 1. 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 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*.

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 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.

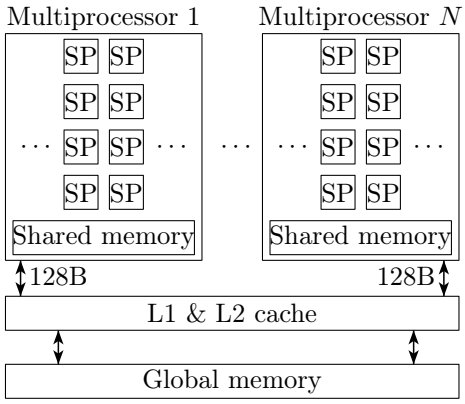

Fig. 1. Hardware model of CUDA GPUs

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 local state (registers and local memory), but their execution is organized in groups of 32 called *warps*. The threads in a warp execute instructions in a synchronous manner, meaning that they move through the code in lockstep. 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.

3.2 Related GPU Implementations

Sparse graphs are usually stored in the *Compressed Sparse Row* 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 $\mathit{offsets}[i + 1] - 1$.

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 [27]; first, they start with reading $\mathit{offsets}[\mathit{ThreadId}]$ and $\mathit{offsets}[\mathit{ThreadId} + 1]$, later possibly moving to other offsets depending on the total number of threads running and the size of the graph. Each time that offsets have been read, and the corresponding source state is in the search frontier, the relevant range of transitions can be accessed next, and, in cases that the target states have not yet been visited, these are added to the new frontier.

Li *et al.* [7] 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. [28], 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 [5], 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 [7], 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. [6] 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.

3.3 SCC Decomposition on the GPU

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 CUDA provides special atomic read and write operations for them. 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*. The highest bit of 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 second and third highest 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}

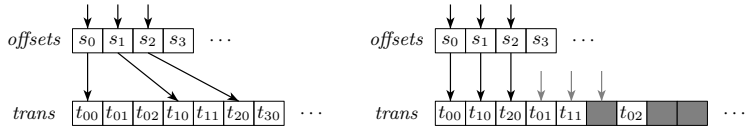

Fig. 2. Fetching transitions before and after restructuring

states, i.e. about 537 million states. For a graph to be decomposed by our GPU implementation, at least $2 \cdot |V| + |E| + 1$ integers are needed. A typical GPU has up to 5 GB global memory, which allows up to 1.3 billion integers to be stored, hence 29 bits is sufficient to refer to all the states of a graph that can be handled.

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 2. For the sake of clarity, we assume in this example that the warp size is 3. 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 2, 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*. 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 across the whole *trans* array.

Algorithm. To illustrate our implementation of FBT for GPUs, we will discuss some of its more interesting aspects. Essentially, every step of Alg. 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.

Alg. 2 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. Note that GPU specific notions such as *NrOfThreads* and *BlockSize* have been defined in

Algorithm 2 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 []
 2: <initialise cache>
    for (i ← Global-ThreadId; i < |V|; i ← i + NrOfThreads) do
 4:     srcinfo ← offsets[i]
        if INFRONTIER(srcinfo) then
 6:         offsets[i] ← MOVETOEXPLORED(srcinfo)
        EXPLORE(srcinfo)
 8: for (iter ← 1; iter < NrIters; iter ++) do
        for i ← ThreadId; i < cachesize; i ← i + BlockSize do
10:         srcinfo ← cache[i]
            if srcinfo ≠ empty then
12:             cache[i] ← empty
                EXPLORE(srcinfo)
```

Section 3.1. Two 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. If a state is in the frontier, it is removed and set to explored by the operation MOVETOEXPLORED at line 6. After that, the state is explored.

This 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. 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, 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 Alg. 3, the GPU 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. If so, the variable *thcont* is set, which at line 7, when all the threads in a warp exchange their value of *thcont* (the BROADCAST procedure), results in the entire warp commencing with the exploration, since at least one thread needs to explore. At line 9, *srcregion* stores the FBT search region (see Section 2.2) to which state i belongs. Note that at lines 11-12, if the thread is scanning a segment, the upper bound offset can be derived using the segment interval, and reading a second *offsets* entry can actually be avoided. The segment interval indicates the number of transition entries for each source state in a segment. Starting at line 15, the successors of i are read. Threads that are only reading entries to assure coalesced accesses do not execute lines beyond line 17. At line 20, ISACTIVE checks if the search region of the target state j 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 22), where *tgtregion* represents the search region of the target state (line 21), then the target state is elligible 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 25-27). 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 (lines 30-31). 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 Alg. 3, 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.

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*. Note that state i belongs to search region $results[i]$. For this state, location $results[i] + \text{REACHEDINBWD}(results[i]) + 2 \cdot \text{REACHEDINFWD}(results[i])$, will be accessed in *trans*, with REACHEDINBWD and REACHEDINFWD indicating whether the state has been reached in the backward or forward BFS, respectively. Since this location may actually be beyond the bounds of *results*, pivot selection is performed in several iterations, in each iteration j only considering the regions with a hash between $j \cdot |E|$ and $(j+1) \cdot |E|$. Once a thread has determined the hash h, it will try to 'claim' the corresponding $trans[h]$ entry by atomically writing the ID of its state with the highest bit set to lock the entry. Exactly one thread i will be able to do this, after which that thread will store the original $trans[h]$ entry temporarily in $results[i]$, and all other threads read the new contents of $trans[h]$, and write this new region information into their *results* entries. The enforced data races are used to pseudo-randomly choose pivots. Finally, to revert *trans* back to its original content, after pivot selection, thread i swaps $results[i]$ and the unlocked $trans[h]$. Note that with this approach, SCCs are actually identified by their pivots, and any number of pivots can be selected in parallel.

3.4 MEC Decomposition on the GPU

Our GPU implementation for MEC decomposition is based on the basic algorithm presented in Section 2. 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. Whenever a state i in the SCC with ID *pivot* is identified to be in U, we lock entry $trans[pivot]$ to indicate that this SCC cannot be a MEC, and we mark $results[i]$ for removal.

Algorithm 3 EXPLORE with local caching for GPU

Require: offset entry *srcinfo* of a state i
Ensure: if i is in search frontier, then the successors of i are added to search frontier, and i is moved to the explored set

```
    thcont = 0
 2: if i < 32 · #segments then
        stepsize ← 32
 4:     thcont = 1
    else
 6:     stepsize ← 1
    BROADCAST(thcont)
 8: if thcont then
        srcregion ← GETREGION(results[i])
10:     offset1 ← GETOFFSET(srcinfo)
        if stepsize = 32 then
12:         offset2 ← offset1 + (32 · segmentinterval)
        else
14:         offset2 ← GETOFFSET(offsets[i + 1])
        for (j ← offset1; j < offset2; j ← j + stepsize) do
16:         t ← trans[j]
            if INFRONTIER(srcinfo) then
18:             k ← GETTGTSTATE(t)
                r ← results[j]
20:             if ISACTIVE(r) then
                    tgtregion ← GETREGION(r)
22:                 if srcregion = tgtregion then
                        tgtinfo ← offsets[k]
24:                     if ISNEW(tgtinfo) then
                            if iter < NrIters − 1 then
26:                             if ¬STOREINCACHE(k) then
                                    nextIter ← true
28:                         else
                                nextIter ← true
30:                         ADDTOFRONTIER(tgtinfo)
                            offsets[k] ← tgtinfo
```

After that, 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 *pivot* and *trans*[*pivot*] 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 a defined 'empty' value (and their *offsets* entry is locked), and discovered MECs are locked as well. Locking of *offsets* and *results* entries means that the highest bit is set, and those entries are effectively removed from the search.

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.

4 Experiments

We conducted experiments to measure the performance of our implementations using a representative set of benchmark models taken from the standard distribution of the PRISM model checker and additional models provided through

Table 1. SCC decomposition results of Tarjan and several GPU configurations

Model	|V|	|E|	out	#CC	Tar	F0,1	F0,7	F3,1	F3,7	F0,1-nh
wlan.2500	12.6M	28.1M	2.23	12.5M	**6.17**	29.16	20.71	119.00	61.90	26.73
phil.7	11.0M	98.5M	8.97	1	23.47	**0.70**	0.71	1.14	1.18	0.73
diningcrypt.t.10.(0.5)	42.9M	279.4M	6.51	42.9M	41.42	1.63	**1.62**	1.74	1.75	2.08
test-and-set.7	51.4M	468.5M	9.12	4672	103.92	30.33	36.04	95.67	92.40	**19.12**
leader.7	68.7M	280.5M	4.08	42.2M	68.08	45.02	**5.35**	110.18	12.27	47.84
phil_lss.5,10	72.9M	425.6M	5.84	1	99.75	3.28	3.30	6.46	6.34	**3.25**
coin.8.3	87.9M	583.0M	6.63	5.4M	135.61	125.94	**9.10**	582.59	42.04	179.00
mutual.7.13	76.2M	653.7M	8.58	1	121.31	4.08	**3.72**	4.97	4.66	4.71
zeroconf_dl.F.200.1k.6	118.6M	273.5M	2.31	118.6M	97.91	28.63	**6.12**	28.98	6.23	28.63
firewire_dl.800.36.(0.2)	129.3M	293.6M	2.27	129.3M	104.07	26.71	**6.71**	26.97	6.87	26.60

its dedicated website.[5] 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 2496 cores and 5 GB global memory.

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.

For comparison, we used a CPU implementation of Tarjan's SCC decomposition. Table 1 presents the graph characteristics of the cases and the runtimes in seconds running the CPU and GPU implementions, the latter in a range of different configurations. The 'out' column provides the average out-degree, while the '#CC' column displays the number of SCCs in the graph. 'Tar' stands for Tarjan, and Fi,j represents GPU FBT with i search iterations per BFS kernel launch using the local cache, and j being the interval (out-degree upperbound) used for restructuring the input. Finally, F0, 1-*nh* is an FBT search in which we have disabled the hybrid search kernel.

Table 2. MEC dec. results

Model	BM	GM
wlan.2500	32.33	21.46
phil.7	51.22	0.73
diningcrypt.t.10.(0.5)	140.85	1.80
test-and-set.7	203.50	36.70
leader.7	239.80	7.48
phil_lss.5,10	281.32	3.45
coin.8.3	363.07	12.63
mutual.7.13 -N	302.66	3.83
zeroconf_dl.F.200.1k.6	390.05	6.23
firewire_dl.800.36.(0.2)	470.96	6.90

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.

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

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

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.

Concerning the latter two cases and other 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 [5], 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 [5] 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. The controlled experiments in which we disabled the hybrid search kernel (F0,1-*nh*) shows that using the 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 some cases using them causes a slowdown. 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 Table 2, results for MEC decomposition are presented. BM stands for Basic MEC decomposition on the CPU, using Tarjan's SCC decomposition for the first step. GM is GPU MEC decomposition using the overall best setup without caches and with restructuring (F0,7). Speedups up to 79 times were measured. The cause for the increased speedups is that the additional steps after SCC decomposition in BM can be performed extremely efficient in parallel on a GPU, since they require (fully coalesced) scanning of the input arrays.

5 Conclusions

We presented GPU algorithms for finding SCCs and MECs in sparse graphs. 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. The extra steps for MEC decomposition are very suitable for GPUs.

For future work, we plan to address similar problems in probabilistic model checking [1], and to integrate the algorithms in model checking tools.

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
2. Wang, C., Bloem, R., Hachtel, G.D., Ravi, K., Somenzi, F.: Compositional SCC analysis for language emptiness. Formal Methods in System Design **28** (2006) 5–36
3. Tarjan, R.: Depth-First Search and Linear Graph Algorithms. SIAM J. Comput. **1** (1972) 146–160
4. Bloem, R., Gabow, H.N., Somenzi, F.: An Algorithm for Strongly Connected Component Analysis in $n \log n$ Symbolic Steps. Formal Methods in System Design **28** (2006) 37–56
5. Barnat, J., Bauch, P., Brim, L., Ceska, M.: Computing Strongly Connected Components in Parallel on CUDA. In: IPDPS, IEEE (2011) 544–555
6. Hong, S., Rodia, N., Olukotun, K.: On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-World Graphs. In: SC'13, ACM (2013) 92
7. Li, G., Zhu, Z., Cong, Z., Yang, F.: Efficient Decomposition of Strongly Connected Components on GPUs. Journal of Systems Architecture **60** (2014) 1–10
8. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM **42** (1995) 857–907
9. de Alfaro, L.: How to specify and verify the long-run average behavior of probabilistic systems. In: LICS, IEEE Computer Society (1998) 454–465
10. Ummels, M., Wojtczak, D.: The Complexity of Nash Equilibria in Stochastic Multiplayer Games. Logical Methods in Computer Science **7** (2011)
11. Bruyère, V., Filiot, E., Randour, M., Raskin, J.F.: Meet your expectations with guarantees: Beyond worst-case synthesis in quantitative games. In: STACS. Volume 25 of LIPIcs., Schloss Dagstuhl (2014) 199–213
12. Chatterjee, K., Henzinger, M.: Faster and Dynamic Algorithms for Maximal End-Component Decomposition and Related Graph Problems in Probabilistic Verification. In: SODA, SIAM (2011) 1318–1336
13. Chatterjee, K., Łącki, J.: Faster Algorithms for Markov Decision Processes with Low Treewidth. In: CAV. Volume 8044S of LNCS., Springer (2013) 543–558
14. Fleischer, L., Hendrickson, B., Pinar, A.: On Identifying Strongly Connected Components in Parallel. In: IPDPS Workshop Irregular 2000. Volume 1800 of LNCS., Springer (2000) 505–511
15. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-time Systems. In: CAV. Volume 6806 of LNCS., Springer (2011) 585–591
16. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: Parallel Probabilistic Model Checking on General Purpose Graphic Processors. STTT **13** (2011) 21–35
17. Barnat, J., Brim, L., Ceska, M., Lamr, T.: CUDA Accelerated LTL Model Checking. In: ICPADS, IEEE (2009) 34–41
18. Edelkamp, S., Sulewski, D.: Efficient Explicit-State Model Checking on General Purpose Graphics Processors. In: SPIN. Volume 6349 of LNCS., Springer (2010) 106–123
19. Wijs, A., Bošnački, D.: Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking. In: SPIN. Volume 7385 of LNCS., Springer (2012) 98–116
20. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: GPU-PRISM: An Extension of PRISM for General Purpose Graphics Processing Units. In: PDMC 2010, IEEE (2010) 17–19
21. Wijs, A., Bošnački, D.: GPUexplore: Many-Core On-The-Fly State Space Exploration. In: TACAS. Volume 8413 of LNCS., Springer (2014) 233–247

22. Chatterjee, K., Henzinger, M.: An $O(n^2)$ Time Algorithm for Alternating Büchi Games. In: SODA, SIAM (2012) 1386–1399
23. Dijkstra, E.W., Feijen, W.H.J.: A Method of Programming. Addison-Wesley (1988)
24. McLendon III, W., Hendrickson, B., Plimpton, S., Rauchwerger, L.: Finding Strongly Connected Components in Distributed Graphs. J. Parallel Distrib. Comput. **65** (2005) 901–910
25. Orzan, S.: On Distributed Verification and Verified Distribution. PhD thesis, Free University of Amsterdam (2004)
26. Barnat, J., Moravec, P.: Parallel Algorithms for Finding SCCs in Implicitly Given Graphs. In: FMICS/PDMC. Volume 4346 of LNCS., Springer (2007) 316–330
27. Harish, P., Narayanan, P.: Accelerating Large Graph Algorithms on the GPU Using CUDA. In: HiPC. Volume 4873 of LNCS., Springer (2007) 197–208
28. Merrill, D., Garland, M., Grimshaw, A.: Scalable GPU Graph Traversal. In: PPoPP, ACM (2012) 117–128
29. Stuhl, M.: Computing Strongly Connected Components with CUDA. Master's thesis, Masaryk University (2013)