GPU Accelerated Strong and Branching Bisimilarity Checking

Anton Wijs[1,2,⋆]

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

Abstract. Bisimilarity checking is an important operation to perform explicit-state model checking when the state space of a model under verification has already been generated. It can be applied in various ways: reduction of a state space w.r.t. a particular flavour of bisimilarity, or checking that two given state spaces are bisimilar. Bisimilarity checking is a computationally intensive task, and over the years, several algorithms have been presented, both sequential, i.e. single-threaded, and parallel, the latter either relying on shared memory or message-passing. In this work, we first present a novel way to check strong bisimilarity on general-purpose graphics processing units (GPUs), and show experimentally that an implementation of it for CUDA-enabled GPUs is competitive with other parallel techniques that run either on a GPU or use message-passing on a multi-core system. Building on this, we propose, to the best of our knowledge, the first many-core branching bisimilarity checking algorithm, an implementation of which shows speedups comparable to our strong bisimilarity checking approach.

1 Introduction

Model checking [2] is a formal verification technique to ensure that a model satisfies desired functional properties. There are essentially two ways to perform it; *on-the-fly*, which means that properties are being checked while the model is being analysed, i.e. while its state space is explored, and *offline*, in which first the state space is fully generated and subsequently properties are checked on it. For the latter case, it is desirable to be able to compare and minimise state spaces, to allow for faster property checking. In action-based model checking, Labelled Transition Systems (LTS*s*) are often used to formalise state spaces, and (some flavour of) bisimilarity is used to compare and minimise them. Checking bisimilarity of LTS*s* is a computationally intensive operation, and over the years, several algorithms have been proposed, e.g. [17,20,15,19].

Graphics Processing Units (GPUs) have been used in recent years to dramatically speed up computations. For model checking, algorithms have been

⋆ 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).

C. Baier and C. Tinelli (Eds.): TACAS 2015, LNCS 9035, pp. 368–383, 2015.
DOI: 10.1007/978-3-662-46680-3_29

presented to use GPUs for several critical operations, such as on-the-fly state space exploration [4,23], offline property checking [3,8,9,22], counterexample generation [27], state space decomposition [24], but strong bisimilarity checking has not received much attention, and branching bisimilarity [14] has received none. In this paper, we propose new algorithms for these operations, in the latter case only assuming that the LTSs do not contain cycles of internal behaviour.

Structure of the paper. In Section 2, we present the basic notions used in this paper. Section 3 contains a discussion of the typical GPU setting, and explains how to encode the required input. In Section 4, we present our new algorithms, and Section 5 contains our experimental results. Finally, related work is discussed in Section 6, and Conclusions are drawn in Section 7.

2 Preliminaries

In this section, we discuss the basic notions involved to understand the problem, namely labelled transition systems, strong and branching bisimilarity, and the existing basic approaches to check strong and branching bisimilarity.

Labelled Transition Systems. We use Labelled Transition Systems (LTSs) to represent the semantics of finite-state systems. They are action-based descriptions, indicating how a system can change state by performing particular actions. An LTS $\mathcal{G}$ is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \underline{s} \rangle$, where $\mathcal{S}$ is a (finite) set of states, $\mathcal{A}$ is a set of actions or labels (including the invisible action τ), $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is a transition relation, and $\underline{s} \in \mathcal{S}$ is the initial state. Actions in $\mathcal{A}$ are denoted by a, b, c, etc. We use $s_1 \xrightarrow{a} s_2$ to denote $\langle s_1, a, s_2 \rangle \in \mathcal{T}$. If $s_1 \xrightarrow{a} s_2$, this means that in $\mathcal{G}$, an action a can be performed in state s_1, leading to state s_2. With $\mathcal{T}(s)$, we refer to the set of states that can be reached by following a single outgoing transition of s. Finally, the special action τ is used to denote internal behaviour of the system, and $s_1 \Rightarrow s_2$ indicates that it is possible to move from s_1 to s_2 via 0 or more τ-transition, i.e. $\Rightarrow$ is the reflexive, transitive closure of $\xrightarrow{\tau}$.

Strong Bisimilarity. The first equivalence relation between LTSs that we consider in this paper is strong bisimilarity.

Definition 1 (Strong Bisimulation). *A binary relation $R \subseteq \mathcal{S} \times \mathcal{S}$ is a* strong bisimulation *if R is symmetric and $s\ R\ t$ implies that if $s \xrightarrow{a} s'$ then $t \xrightarrow{a} t'$ with $s' R\ t'$.*

Two states s and t are *bisimilar*, denoted by $s \leftrightarrows t$, if there is a strong bisimulation relation R such that $s\ R\ t$.

In this paper, when trying to construct a bisimulation relation, we are always interested in the *largest* bisimulation. Strong bisimilarity is closed under arbitrary union, so this largest relation is the combination of all relations that can be constructed.

The problem of checking strong bisimilarity for LTSs when $|\mathcal{A}| = 1$ corresponds with the single function coarsest partition problem. The most widely known algorithms to solve this problem is by Paige & Tarjan (PT) [20] and

by Kanellakis & Smolka (KS) [17], and both can be extended for the multiple functions coarsest partition problem, to handle LTSs with multiple actions.

A bisimilarity checking algorithm can be used both to minimise an LTS, by reducing all bisimilar states to a single state in the output LTS, and to compare two LTSs $\mathcal{G}_1 = \langle \mathcal{S}_1, \mathcal{A}_1, \mathcal{T}_1, \underline{s}_1 \rangle$, $\mathcal{G}_2 = \langle \mathcal{S}_2, \mathcal{A}_2, \mathcal{T}_2, \underline{s}_2 \rangle$. The latter boils down to checking whether $\underline{s}_1$ and $\underline{s}_2$ end up being bisimilar after checking bisimilarity on the combined LTS $\langle \mathcal{S}_1 \cup \mathcal{S}_2, \mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{T}_1 \cup \mathcal{T}_2, \underline{s}_1 \rangle$ (for convenience, we assume that $\mathcal{S}_1 \cap \mathcal{S}_2 = \emptyset$).

We proceed with explaining the basic mechanism to check bisimilarity that we use in the remainder of this paper, which is partition refinement. In fact, this mechanism is the so-called "naive" reduction algorithm[1] mentioned by Kanellakis & Smolka [17], since it has been shown in the past to be suitable for parallelisation and, unlike PT, it can be extended straightforwardly for branching and weak bisimilarity [19]. We further motivate the use of this mechanism in Section 3 after the explanation of the GPU basics.

A *partition* of $\mathcal{S}$ is a set of m disjoint state sets called *blocks* B_i $(1 \leq i \leq m)$ such that $\bigcup_{1 \leq i \leq m} B_i = \mathcal{S}$. A partition refinement algorithm takes as input a partition, analyses it, and produces as output a *refinement* of that partition. A partition π' is a refinement of π iff every block of π' is contained in a block of π.

The idea behind using partition refinement for bisimilarity checking is that initially, a partition π consisting of a single block $B = \mathcal{S}$ is defined, which is then further refined on the criterion whether states in the same block can be distinguished w.r.t. π until no further refining can be done. The resulting partition then represents a bisimulation relation: two states s, t are bisimilar iff they are in the same block.

The problem of checking strong bisimilarity of LTSs with multiple transition labels, i.e. the multi-function coarsest partition problem, can now be formalised as follows:

Definition 2 (Strong Bisimilarity Checking Problem). *Given an* LTS $\mathcal{G} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \underline{s} \rangle$ *and an initial partition* $\pi_0 = \{\mathcal{S}\}$*, find a partition* π *such that:*

1. $\forall B \in \pi,\ s,t \in B,\ a \in \mathcal{A},\ B' \in \pi.\ (\exists s' \in B'.s \xrightarrow{a} s' \iff \exists t' \in B'.t \xrightarrow{a} t')$*;*
2. *No partition* $\pi' \neq \pi$ *can be constructed which refines* π *and satisfies 1.*

Blom et al. [5,6,7] and Orzan [19] define the notion of a *signature* of a state, to reason about condition 1 in Def. 2. The signature $sig_\pi(s)$ of a state s in a partition π encodes which transitions can be taken from s and to which blocks in π they lead. In the following definition of $sig_\pi(s)$, we interpret a partition π as a function $\pi : \mathcal{S} \to \mathbb{N}$:

$$sig_\pi(s) = \{(a, \pi(s')) \mid s \xrightarrow{a} s'\}$$

In each iteration of a partition refinement algorithm, we can now check for each block $B \in \pi$ and each two states $s, t \in B$ whether $sig_\pi(s) = sig_\pi(t)$. If so, then

[1] In [17], some optimisations on this algorithm are presented. How well these are applicable in a GPU setting remains to be investigated.

Algorithm 1. Partition refinement with signatures

Require: $\mathcal{G} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \underline{s} \rangle$, $\pi = \{\mathcal{S}\}$
  $stable \leftarrow$ **false**
2: **while** $\neg stable$ **do**
    for all $B \in \pi$ **do**
4: $\pi' \leftarrow (\pi \setminus \{B\}) \cup \{B_1, \ldots, B_m\}$,
      with $\bigcup_{1 \le i \le m} B_i = B \wedge \forall 1 \le i, j \le m. \forall s \in B_i, t \in B_j. (i = j \iff sig_\pi(s) = sig_\pi(t))$
6: **if** $\pi \neq \pi'$ **then**
      $\pi \leftarrow \pi'$
8: **else**
      $stable \leftarrow$ **true**

they should remain in the same block; if not, then B needs to be split. See Alg. 1 for this procedure.

Branching Bisimilarity. The second relation we consider is *branching bisimilarity* [14]. It is sensitive to internal behaviour while preserving the branching structure of an LTS, meaning that it preserves the potential to perform actions, even when internal behaviour is involved. It has several nice properties, among which are the facts that temporal logics such as the Hennessy-Milner logic with an until operator and CTL*-X characterise it [12].

Definition 3 (Branching Bisimulation). *A binary relation $R \subseteq \mathcal{S} \times \mathcal{S}$ is a* branching bisimulation *if R is symmetric and $s\ R\ t$ implies that if $s \xrightarrow{a} s'$ then*
- *either $a = \tau$ with $s' R\ t$;*
- *or $t \Rightarrow \hat{t} \xrightarrow{a} t'$ with $s\ R\ \hat{t}$ and $s' R\ t'$.*

Two states s and t are *branching bisimilar*, denoted by $s \leftrightarrows_b t$, if there is a branching bisimulation R such that $s\ R\ t$. Again, as in the case for strong bisimilarity, we are interested in the *largest* branching bisimulation when checking branching bisimilarity in an LTS.

A well-known property of branching bisimilarity is called *stuttering*, which plays an important role when constructing an algorithm to check branching bisimilarity of LTS*s*:

Definition 4 (Stuttering [14]). *Let R be the largest branching bisimulation relating states in $\mathcal{G} = \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \underline{s} \rangle$. If $s \xrightarrow{\tau} s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_n \xrightarrow{\tau} s'$ ($n \ge 0$) is a path such that there is a $t \in \mathcal{S}$ with $s\ R\ t$ and $s' R\ t$, then for all $1 \le i \le n$, we have $s_i\ R\ t$.*

Def. 4 defines the notion of an *inert τ-path*, or inert path, in which all intermediate states are branching bisimilar with each other. Alg. 1 can in principle be used directly for checking branching bisimilarity if we redefine $sig_\pi(s)$ as $sig^b_\pi(s)$, where $s \overset{\pi}{\Rightarrow} \hat{s}$ expresses that there exists a τ-path between states s, $\hat{s}$ which is inert w.r.t. π:

$$sig^b_\pi(s) = \{(a, \pi(s')) \mid \exists \hat{s} \in \pi(s). s \overset{\pi}{\Rightarrow} \hat{s} \xrightarrow{a} s' \wedge (a \neq \tau \vee \pi(s) \neq \pi(s'))\}$$

In the case of branching bisimilarity, τ-transitions are either *inert* (or *silent*) or not, depending on whether following the transitions results in losing potential

behaviour. This defines whether the source and target states of a τ-transition are branching bisimilar or not. Consider the LTS shown in Fig. 2. From s_1, a τ-transition to s_7 can be done, in which we have a c-loop, and a τ-transition to state s_3. The latter transition is inert, since also in s_3, a τ-transition can be done to a state, s_6, which is branching bisimilar to s_7. In other words, s_3 can *simulate* the behaviour of s_1. However, in line with the stuttering property, inertness applies to transitive closures of τ-transitions. In the example, also $s_0 \xrightarrow{s}_1$ is inert, since the a-transition from s_0 can be simulated by s_3. Hence, we have $s_0 \leftrightarrows_b s_1 \leftrightarrows_b s_3$.

The definition of $sig^b_\pi(s)$ actually refers to π-*inertness* which means that both the source and target state of a τ-transition are in the same block in π. The added complication when checking branching bisimilarity w.r.t. strong is hence that closures of τ-transitions that are π-inert need to be taken into account. Because of this, the problem of checking branching bisimilarity is also known as the multiple functions coarsest partition with stuttering problem.

In the algorithm by Browne et al. [10] for checking stuttering equivalence, in every iteration, it needs to be checked whether for two states s and t the behaviour reachable via inert paths is equivalent, in order to establish that s and t are equivalent. This means that for each pair of possibly equivalent states, inert paths need to be reexplored. The complexity of the algorithm is $O(|\mathcal{S}|^5)$.

In the algorithm by Groote & Vaandrager (GV) [15], reexploration of inert paths is avoided, and its complexity is $O(|\mathcal{S}| \cdot (|\mathcal{S}| + |\mathcal{T}|))$. There, a pair of blocks (B, B') must be identified such that there both is a state in B with a transition to B', and there is no bottom state in B with a transition to B'. A state in B is a bottom state when it has no transition to a state in B. If such a pair of blocks can be found, then B must be split. This splitting criterion is directly based on the previously mentioned observation that a τ-path is inert iff it leads to a state which can simulate all behaviour of the intermediate states. Because of this, GV requires that no τ-cycles are present. This is not an important restriction, since compressing τ-cycles into individual states can be done in $O(|\mathcal{T}|)$ [1].

After the next section, explaining the basics of GPUs, we return to checking bisimilarity, focussing on existing approaches for many-core settings, and motivating our approach.

3 GPU Basics

In this paper, we focus on NVIDIA GPU architectures and the Compute Unified Device Architecture (CUDA) interface. However, our algorithms can be straightforwardly applied to any architecture with massive hardware multithreading and the SIMT (Single Instruction Multiple Threads) model.

CUDA is NVIDIA's interface to program GPUs. It extends C and FORTRAN. We use the C extension. CUDA includes special declarations to explicitly place variables in the various types of memory (see Figure 1), predefined keywords to refer to the IDs of individual threads and blocks of threads, synchronisation statements, a run time API for memory management, and statements to define

and launch GPU functions, known as *kernels*. In this section we give a brief overview of CUDA. More details can be found in, for instance, [9,23].

CUDA Programming Model. A CUDA program consists of a *host* program running on the Central Processing Unit (CPU) and a (collection of) CUDA kernels. Kernels describe the parallel parts of the program and are executed many times in parallel by different threads on the GPU device. They are launched from the host. Often at most one kernel can be launched at a time, but there are also GPUs that allow running multiple different kernels concurrently. 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. A block of threads is assigned to a multiprocessor.

CUDA Memory Model. Threads have access to different kinds of memory. Each thread has a number of on-chip registers that allow fast access. Furthermore, threads within a block can together use the *shared memory* of a multiprocessor, which is also on-chip and fast. Finally, all blocks have access to the *global memory* which is large (currently up to 12 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, which allows it to be used for communication between the host and the kernel.

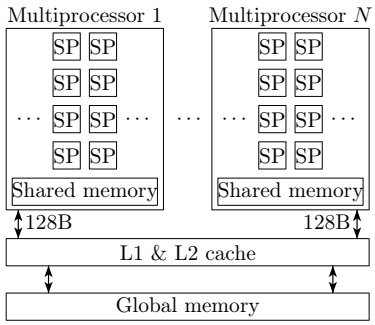

Fig. 1. Hardware model of CUDA GPUs

GPU Architecture. A GPU contains a set of streaming multiprocessors (SMs), and each of those contains a set of SPs. The NVIDIA Kepler K20m, which we used for our experiments, has 13 SMs, each having 192 SPs, which is 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 organised in groups of 32 called *warps*. The threads in a warp execute instructions in lock-step, i.e. they share a program counter. If the memory accesses of threads in a warp can be grouped together physically, i.e. if the accesses are coalesced, then the data can be obtained using a single fetch, which greatly improves the runtime compared to fetching physically separate data. When checking bisimilarity on state spaces, though, the required access to transitions is expected to be

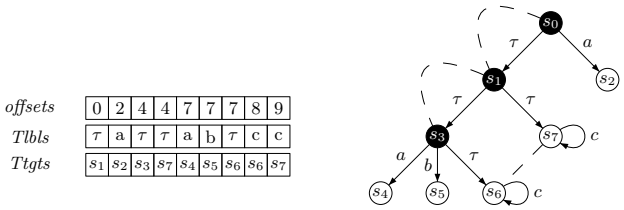

Fig. 2. An example LTS and its encoding in *offsets*, *Tlbls* and *Ttgts* arrays

irregular. This poses the challenge of reducing the number of irregular memory accesses despite of that fact.

LTS *Representation.* In order to check bisimilarity of LTS*s* on a GPU, we first need to find a suitable encoding of them to store the input data in the global memory. For this, we use a representation similar to those used to compactly describe sparse graphs. Fig. 2 shows an example of such an encoding of the LTS on the right. Three arrays are used to store the information. The first one, *offsets*, holds for every state i the start and end indices of its outgoing transitions in the other two arrays at *offsets*$[i]$ and *offsets*$[i+1]$, respectively. Arrays *Tlbls* and *Ttgts* provide a list of the outgoing transitions, in particular their action labels and target states, respectively. In practice, actions are encoded by an integer, with $\tau = 0$, $a = 1$, etc. To give an example, the transitions of s_1 can be found from *offsets*$[1]$ up to *offsets*$[2]$, i.e. at positions 2 and 3, in *Tlbls* and *Ttgts*. Finally, it should be noted that the outgoing transitions of each state have been sorted by label lexicographically, with τ the smallest element. We will use this to our advantage later on, when we explain our multi-way splitting procedure.

Finally, we note that in the following, when we refer to an array entry as being *locked*, we mean that its highest bit has been set. In general, we use 32-bit integers to store data elements. Even if we reserve the highest bit of each entry, we can still refer to 2^{31} states. In a connected LTS, this means that we will also have at least $2^{31}-1$ transitions. Since transitions take two integers to store each (in *Tlbls* and *Ttgts*), an LTS of that size would not fit in current GPUs anyway.

4 Many-Core Bisimilarity Checking

Strong Bisimilarity. The algorithm by Lee & Rajasekaran (LR) [18] is the first that has been proposed to check strong bisimilarity on SIMT architectures, and is based on KS. We discuss the main approach of it here since we will justify the choices we made for our algorithm w.r.t. LR, and since we have an experimental comparison between CUDA-implementations of LR (made by us) and the new algorithm in Section 5.

Table 1 shows an example situation when running LR on the LTS in Fig. 2. A number of arrays are used here. First of all, not listed in the table, a B

Table 1. Running LR on the LTS in Figure 2

P	$(0,s_0)$	$(0,s_3)$	$(0,s_4)$	$(1,s_1)$	$(1,s_6)$	$(2,s_2)$	$(2,s_5)$	$(2,s_7)$	-
V	0	1	2	0	1	0	1	2	-
TSIZE	2	3	0	2	1	0	0	1	-
Lsrc	$B[s_0]$	$B[s_0]$	$B[s_3]$	$B[s_3]$	$B[s_3]$	$B[s_1]$	$B[s_1]$	$B[s_6]$	$B[s_7]$
Llbl	τ	a	τ	a	b	τ	τ	c	c
Lidx	0	0	1	1	1	0	0	1	2
Ltgt	$B[s_1]$	$B[s_2]$	$B[s_6]$	$B[s_4]$	$B[s_5]$	$B[s_3]$	$B[s_7]$	$B[s_6]$	$B[s_7]$

array is maintained indicating to which block each state i belongs. With this, a partition array P is constructed consisting of tuples $(B[i], i)$, which is then sorted in parallel on $B[i]$ (initially, if we have a single block, no sorting is required). Next, a V array is filled assigning to each state i a block local ID between 0 and $|B[i]|$, i.e. an identifier local to the block it is in, which can be done using P. The order in which the states appear in P determines array *TSIZE*; the latter must be filled with the number of outgoing transitions of each corresponding state in P, which can be done in parallel using P and *offsets*. Finally, after obtaining absolute offsets using *TSIZE*, the L arrays are filled, listing the transitions in the LTS w.r.t. the current partition in the order of the states in P. Besides source and target block and the label, also the block local ID of the source is added in $Lidx$. Note that from L we can now directly learn the signature of each state.

Once L is filled, it is lexicographically sorted. Because the block local IDs have been included, this means that all transitions of a state are still next to each other, but now also sorted by label and target block. After removing duplicate entries in parallel, we have essentially made sure that the lists of outgoing transitions can be interpreted as sets. Then, the most interesting operation is performed, namely the comparison of signatures. For this, LR compares in parallel the signature of each state i with the one of state $i - V[i]$, i.e. of the state with block local ID 0 of the same block. This signature can directly be found using V. When the signatures are equal, nothing is done, but when they are not, a new block ID x is created and state i is assigned to it, i.e. $B[i]$ is set to x. How new block IDs should be chosen in parallel is not mentioned in [18], in fact, they split blocks sequentially, but we chose for the following mechanism: threads working on the same block and finding states that must be split off to select a new block ID first check whether $TSIZE[i - V[i]]$ is locked, i.e. whether its highest bit is set, and if not, lock it atomically. Only one thread will succeed in doing so, which will subsequently try to atomically increment a global ID counter. Once it succeeds, it stores the new value in $TSIZE[i - V[i]]$. After that, the other threads read this value and learn the new block ID.

LR is a typical SIMT application; all data is stored in arrays, and the threads manipulate these on a one-on-one basis, i.e. n threads work on arrays of size n. Moreover, LR uses a number of parallel operations, such as sorting and performing segmented scans, that are available in standard GPU libraries. For instance, we have implemented LR using the THRUST library. However, we chose to design an algorithm which is very different from this one, based on the following ideas:

1. Comparing states with a specific 'first' state is very suitable for a GPU, since it allows for threads to check locally whether their state needs to move to

another block, but we observe that this can be any state, and found a way to select such a first state, which we from now on will call a *representative*, without sorting. This allows for more coalesced memory access when threads can be assigned to consecutive states as opposed to consecutive elements of P. The order of states in P can be considered random.

2. Maintaining L requires many (expensive) memory accesses, and the involved reads and writes are not coalesced, due to the randomness imposed by the structure of the LTS. We chose to directly have each thread use the information from *Tlbls*, *Ttgts* and B concerning the involved transitions of both its assigned state and the associated representative, and construct the signatures in its local registers, which allows for fast comparisons, and reduces the memory requirements from $6 \cdot |\mathcal{S}| + 8 \cdot |\mathcal{T}|$ to $2 \cdot |\mathcal{S}| + 2 \cdot |\mathcal{T}|$.
3. If we start with a single block, which is not considered in [18], then each iteration except the last one produces exactly one new block. This does not scale well. In [16], a multi-way splitting procedure is proposed, but it is based on the entire signature of states, and not very suitable for our representative selection. We propose a multi-way splitting mechanism that is compatible.

The new algorithm. First, we will explain multi-way representative selection. In Fig. 3, part of the initial situation is shown when applying our algorithm on the LTS in Fig. 2. Initially, states are not in any block, indicated by $B[i] =$ '-'.

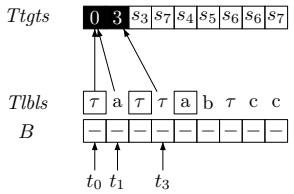

Fig. 3. Representative selection

Each thread t_i checks if state s_i needs to move to a new block, which is indicated by the highest bit in $\mathit{offsets}[i]$ being set. Then, they read a predetermined outgoing transition label of s_i. This is handled globally using variable *labelcounter*; if $\mathit{labelcounter} = j$, this means that every thread reads the label of the j-th transition. Let us call this label ℓ. Whenever $j > |\mathcal{T}(s_i)|$, we say that $\ell = 0$. Using $B[i]$ (interpreting '-' as 0) and ℓ, t_i computes a hash value $h = ((B[i] \cdot N) + \ell) \bmod |\mathcal{T}|$, with N the number of different labels in the LTS. Next, *Ttgts* is temporarily reused to serve as a hash table, and thread i tries to atomically write a locked i to cell $\mathit{Ttgts}[h]$. Only one thread will succeed in doing so per cell. The one that does has now successfully promoted its state to representative of a new block, and the other threads, knowing that they have failed since they encountered a locked entry, read the representative ID from $\mathit{Ttgts}[h]$ and store it in their B cells. Note that in general, h can be larger than $|\mathit{Ttgts}|$. Since we want no threads to meet in *Ttgts* that do not represent states from the same block, the selection procedure is actually done in several iterations, shifting the range of the hash table each iteration by $|\mathit{Ttgts}|$. This way of selecting allows using state IDs as block IDs, which is possible since the blocks are disjoint.

The selection procedure uses a form of multi-way splitting, i.e. splitting a block at once into more than two blocks, when possible, by ensuring that threads which encounter different transition labels try to atomically write to different cells in *Ttgts*. Note that in Section 3, we mentioned that the outgoing transitions of

each state are in *Tlbls* sorted by label. This means that initially, if two states do not have the same first label, then they cannot be bisimilar, and can hence immediately be moved to different blocks. In the next iteration, we know that in each block B_i, all the states must have the same label on their first transition, so we focus on the second transition, and so on. Hence, in order for this to be effective, we use *labelcounter* to change the splitting criterion, which works as long as we have not reached the end of the transition list of at least one state.

Algorithm 2 Many-core bisimilarity checking

```
Require: 𝒢 = ⟨𝒮, 𝒜, 𝒯, s⟩, π = {𝒮}
    stable ← false
 2: while ¬stable do
        labelcounter ← (labelcounter + 1)  mod |𝒯|
 4:     device_stable ← true
        selectRepresentatives ⋘ |𝒮| ⋙ (labelcounter)
 6:     postprocessElection ⋘ |𝒯| ⋙ ()
        while continue do
 8:         device_continue ← false
            propagateBlockIDs ⋘ |𝒮| ⋙ ()
10:         continue ← device_continue
        markNoninertTaus()
12:     compareSignatures ⋘ |𝒮| ⋙ ()
        stable ← device_stable
```

Alg. 2 without the boxed code presents an overview of the entire strong bisimilarity checking procedure. We use the CUDA notation $\lll n \ggg$ to indicate that n threads execute a given kernel. Once new representatives have been selected at line 5, postprocessing is performed to recreate the original *Ttgts* array. This can be done efficiently, since the elements that were removed during representative selection have been temporarily moved to the B array cells of the new representatives. At line 12, each block of threads fetches a consecutive tile of transitions from the global memory and stores it in the shared memory. Each thread i then does the following:

- It reads the outgoing transitions of s_i from the shared memory;
- It fetches $B[s_i]$ from the global memory;
- It employs its warp to fetch the transitions of $B[s_i]$ in a number of coalesced memory accesses, and stores all these transitions into its local registers;
- It looks up the block IDs of the corresponding target states;
- Finally, $sig_\pi(s_i)$ and $sig_\pi(B[s_i])$ are compared.

This procedure results in the highest bit of $offsets[i]$ being set, and the global variable *device_stable* being set to **false** iff the signatures are not equal. The content of *device_stable* is read by the host at line 13, after which another iteration is started or not, depending on its value.

Branching Bisimilarity. For branching bisimilarity, we need to handle the presence of inert paths, as mentioned in Section 2. Without doing so, Alg. 2 would after line 6 provide an incorrect representation of which blocks can be reached from each state, resulting in the signatures comparison at line 12 going wrong. To check branching bisimilarity, we therefore add a procedure to *propagate* block IDs over τ-transitions which can be considered inert at the current iteration. This is similar to the approach in the algorithm by Blom & Van de Pol [7].[2]

By definition, τ-transitions are inert w.r.t. a partition π iff their source and target states are in the same block. However, if we want threads to locally compare their state with the corresponding block representative, without looking beyond

[2] An alternative would be to try to port GV to GPUs, but GV requires several linked lists, and therefore dynamic memory allocation, making it less suitable for GPUs.

their outgoing transitions (which would lead to expensive searching through the global memory), then it cannot be ensured that source and target states are always in the same block. For example, consider the LTS in Fig. 2. Say we perform the initial representative selection without multiway splitting, and states s_1 and s_3 end up in the same block with s_1 as representative. Then a direct comparison of the two states would reveal that they have unequal signatures, even though transition $s_1 \xrightarrow{\tau} s_3$ is inert. On the other hand, $s_1 \xrightarrow{\tau} s_7$ is not inert, so these cases must be distinguishable by the checking algorithm.

To resolve this, we define the notion of a *visible* signature $\overline{sig}_\pi(s) = \{(a, \pi(s')) \mid s \xrightarrow{a} s' \wedge a \neq \tau\}$, and use the label $\overline{\tau}$ to denote a visible τ-transition. Visible τ-transitions are also included in a visible signature, but initially, all τ-transitions are invisible. This means that in our branching bisimilarity checking algorithm (for an overview, see Alg. 2 with the boxed code), we initially select representatives and compare signatures based on signatures without τ-transitions.

After representative selection, it must be checked whether τ-transitions are *possibly* π-inert, and if they are, the block IDs of their target states should be propagated to their source states, making the latter *propagating states*. The condition for a τ-transition to be possibly π-inert directly corresponds with the observation made in Section 2 that inert τ-paths must end in a state which can simulate all previously potential behaviour. The following is a relevant lemma.

Lemma 1. *A state can reach at most one block via π-inert τ-paths.*

Proof. Say that from a state s two blocks B_1, B_2 can be reached via π-inert τ-paths, leading to states $s' \in B_1$, $s'' \in B_2$. Since each subsequently discovered partition will be a refinement of π, we must have that $s' \not\leftrightarrows_b s''$. But then, we must have that $s \leftrightarrows_b s'$ and $s \leftrightarrows_b s''$. From the facts that $s' \not\leftrightarrows_b s''$ and that branching bisimilarity is an equivalence, we derive a contradiction. □

Definition 5 (Inertness condition and propagating state). *A transition $s \xrightarrow{\tau} s'$ is* possibly π-inert*, and s is a* propagating state*, iff*
- *either s' is not propagating, and $(sig_\pi(s) \setminus \{(s, \tau, \pi(s'))\}) \subseteq \overline{sig}_\pi(s')$;*
- *or s' is propagating, and $(sig_\pi(s) \setminus \{(s, \tau, \pi(s'))\}) \subseteq \overline{sig}_\pi(B[s'])$.*

The first alternative in Def. 5 refers to the case that the target state s' is not propagating. In that case, its visible signature should be a superset of $sig_\pi(s)$ minus τ-transitions leading to $\pi(s')$, i.e. s' should be able to simulate s. Note that this uses Lemma 1: all τ-transitions leading to blocks different from $\pi(s')$ in the signature of s are involved in the comparison, since a τ-transition to $\pi(s')$ can only be π-inert if all other τ-transitions to different blocks are not. The second alternative enables propagating results over τ-paths. If s' is propagating, then the signature of the representative $B[s']$ must be taken into account.

In Alg. 2, at lines 7-10, all possibly π-inert τ-transitions are detected, and we mark the source states as propagating using one bit. If a state s has multiple possibly π-inert τ-transitions to different blocks, we define $B[s] =$'-' and mark s propagating. The latter is a conservative action; we refrain from propagating a block ID until we can detect a single block as being reachable via inert τ-paths.

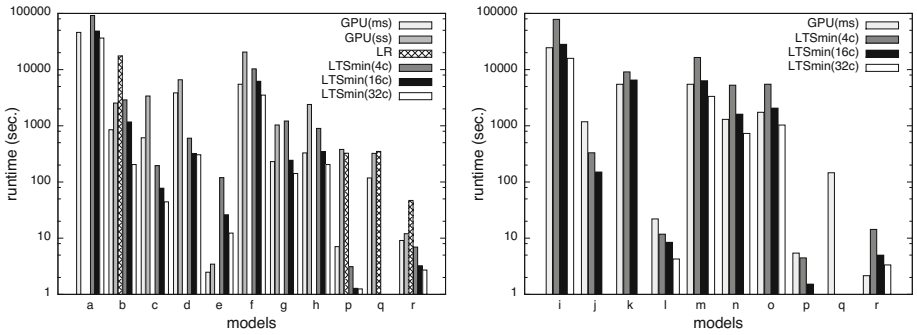

Fig. 4. Strong and branching bisimilarity checking runtimes (sec.)

When propagation finishes, τ-transitions that are not π-inert are relabelled to $\overline{\tau}$ at line 11, thereby adding them to the visible signatures. The signature comparison at line 12 now only concerns the non-propagating states.

Completeness of the algorithm follows from the inertness criterion (Section 2) and the fact that τ-transitions are conservatively marked visible (as long as we do not mark inert τ-transitions as visible, our partition has the largest branching bisimulation as a refinement). Finally, if in any iteration, a state must be moved from '-' to a block, then it must be added to a new block, i.e. one does not need to find an existing suitable block. That this is correct is shown next.

Lemma 2. *Say that state s has been marked non-propagating in iteration $i > 0$, when we have partition π. Then it should be added to a new block not in π.*

Proof. By reasoning towards a contradiction. Consider that when s is marked non-propagating, there already exists a block B' with representative t to which it should be added. Then, in iterations $j < i$, s still had at least one possibly π-inert τ-transition. Let us call the block reachable via that transition B'', and say that $sig_\pi(s) = T \cup \{(\tau, B'')\}$, with T some set of (action, block)-pairs. Because s should be added to B', we must have that t has the same signature as s, hence $sig_\pi(t) = T \cup \{(\tau, B'')\}$. But then, also t must have been propagating in $j < i$, contradicting the assumption that before i, t already represented a block. □

Complexity of the algorithms. If we assume that we can launch $|\mathcal{S}|$ and $|\mathcal{T}|$ threads, then each kernel in Alg. 2 can be performed in O(1). Hence, strong bisimilarity checking can be done in O($|\mathcal{S}|$), since worst-case, $|\mathcal{S}|$ iterations are required. In addition, branching bisimilarity checking requires worst-case a propagation procedure of $|\mathcal{S}|$ iterations, so its complexity is O($|\mathcal{S}|^2$).

5 Experimental Results

In this section, we present some of our experimental comparisons of various GPU setups, and with the LTSmin toolset, which is the only toolset we are aware of

that offers parallel strong and branching bisimilarity checking, and therefore allows us to experimentally compare our implementations with the scalability of CPU approaches.[3] LTSmin offers implementations of the algorithms by Blom et al. [5,6,7]. In this section, we report on a comparison with the algorithms from [5,6]. For the GPU experiments, we used an NVIDIA K20m with 5 GB memory on a machine with an Intel E5-2620 2.0 GHz CPU running CentOS Linux. For the CPU experiments with LTSmin 2.0, we used a machine with an Amd Opteron 6172 processor with 48 cores, 192 GB RAM, running Debian 6.0.7.

Table 2. Benchmark set

id	Model	#st.	#tr.	#ss	#sb
a	BRP250	219m	266m	101m	n.a.
b	coin8.3	87m	583m	20m	n.a.
c	cwi_33949	33m	165m	122k	n.a.
d	cwi_7838	8m	59m	1m	n.a.
e	diningcrypt14	18m	164m	18m	n.a.
f	firewire_dl800.36	129m	294m	34m	n.a.
g	mutualex7.13	76m	654m	76m	n.a.
h	SCSI_C_6	74m	404m	74m	n.a.
i	BRP250 h2	219m	266m	n.a.	19k
j	cwi_33949 h1	33m	165m	n.a.	12k
k	cwi_33949 h2	32m	158m	n.a.	3k
l	diningcrypt14 h2	2m	16m	n.a.	497k
m	firewire_dl800.36 h2	129m	294m	n.a.	26m
n	mutualex7.13 h1	76m	613m	n.a.	41m
o	mutualex7.13 h2	76m	562m	n.a.	32m
p	vasy_6020	6m	19m	7k	256
q	vasy_6120	6m	11m	6k	349
r	vasy_8082	8m	43m	408	290

Table 2 shows the characteristics of each Lts, namely 1) number of states, 2) number of transitions, 3) number of states in the strongly reduced Lts, and 4) number of states in the branching reduced Lts. Note that in some cases, no reduction can be achieved, which is an interesting worst-case scenario for our experiments. The models the Lts*s* stem from have been taken from various sources, namely the Beem database [21], the CADP toolbox [13], the mCRL2 toolset [11], and the website of Prism.[4] To produce cases for branching bisimilarity checking, we wrote a tool to automatically relabel a predefined number of transitions to τ. In the cases suffixed with *h1* roughly 25% of the transitions have the label τ, while in the *h2* Lts*s* this is 50%.

Fig. 4 presents the runtimes we measured (in seconds) for strong and branching bisimilarity checking, on the left and the right, respectively. We used the following setups: a CUDA implementation of our algorithms with (GPU(ms)) and without (GPU(ss)) multi-way splitting, a CUDA-version of LR, and the LTSmin tool ltsmin-reduce-dist running with 4, 16, and 32 cores, which we refer to as LTSmin from now on. In those cases where no result is given for a particular tool and model, the tool ran either out of memory (the CPU tools) or out of time (the GPU tools) on the aforementioned machine.

First of all, considering our algorithm, the positive effect of multiway splitting is apparent, it can speed up the checking by 2 to 50 times, and as expected, since it allows the checking to finish in fewer iterations. Second of all, notice that LR is clearly slower than our approach, in those cases that LR could actually be run, due to its higher memory requirements. These findings support

[3] For a list of all the experiments, and the relevant source code and models, see `http://www.win.tue.nl/~awijs/GPUreduce`

[4] `http://www.prismmodelchecker.org`

the hypothesis that our new approach performs better than LR on modern GPU architectures. In a number of cases, our tool achieves runtimes comparable to the 16-core LTSmin setup. Given that LTSmin scales nicely, this is encouraging. Since the multiway splitting in our approach is more limited to the one in LTSmin, involving a specific transition label as opposed to entire signatures, our tool in particular performs worse when LTSmin can aggressively use multiway splitting. Finally, it is worth noting that sometimes, LTSmin runs out of memory. Storing signatures explicitly is more memory consuming than recreating them every time, but of course, one needs the parallel computation power for the recreation not to be a drawback. Since LTSmin does not exploit shared memory, but keeps the memory separated per worker, this also means that increasing the number of workers tends to increase the presence of redundant information.

6 Related Work

Blom et al. [5,6,7,19] use the aforementioned signatures to distinguish states for distributed bisimilarity checking. In each iteration, the signatures are placed in a hash table, and used as block IDs to refine the partition. On GPUs, however, storing full signatures is very hard, requiring dynamic memory allocation.

Zhang & Smolka [28] propose a parallelisation of KS where threads communicate via message-passing. Such an approach cannot easily be migrated to the GPU setting, since message-passing among threads running on the same GPU does not naturally fit the GPU computation model. On the other hand, one could use message passing between GPUs that together try to check bisimilarity.

Jeong et al. [16] propose a parallel KS algorithm along the lines of [18] with multi-way splitting, but there is a probability that wrong results are obtained.

7 Conclusions

We presented new algorithms to perform strong and branching bisimilarity checking on GPUs. Experiments demonstrate that significant speedups can be achieved. As future work, we will try to further optimise the algorithms. There is still potential to avoid signature comparisons in specific cases. Furthermore, we will consider employing GPUs for other applications of model checking, for instance to find near-optimal schedules (e.g. [26]) and quantitative analysis (e.g. [25]).

References

1. Aho, A., Hopcroft, J., Ullman, J.: The Design and Analysis of Computer Algorithms. Addison-Wesley (1974)
2. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
3. Barnat, J., Bauch, P., Brim, L., Češka, M.: Designing Fast LTL Model Checking Algorithms for Many-Core GPUs. J. Parall. Distrib. Comput. 72, 1083–1097 (2012)
4. Bartocci, E., DeFrancisco, R., Smolka, S.: Towards a GPGPU-parallel SPIN Model Checker. In: SPIN, pp. 87–96. ACM (2014)

5. Blom, S., Orzan, S.: Distributed Branching Bisimulation Reduction of State Spaces. In: FMICS. ENTCS, vol. 80, pp. 109–123. Elsevier (2003)
6. Blom, S., Orzan, S.: A Distributed Algorithm for Strong Bisimulation Reduction of State Spaces. STTT 7(1), 74–86 (2005)
7. Blom, S., van de Pol, J.: Distributed Branching Bisimulation Minimization by Inductive Signatures. In: PDMC. EPTCS, vol. 14, pp. 32–46. Open Publishing Association (2009)
8. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: GPU-PRISM: An Extension of PRISM for General Purpose Graphics Processing Units. In: PDMC 2010, pp. 17–19. IEEE (2010)
9. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: Parallel Probabilistic Model Checking on General Purpose Graphic Processors. STTT 13(1), 21–35 (2011)
10. Browne, M., Clarke, E.M., Grumberg, O.: Characterizing Finite Kripke Structures in Propositional Temporal Logic. TCS 59, 115–131 (1988)
11. Cranen, S., Groote, J.F., Keiren, J.J.A., Stappers, F.P.M., de Vink, E.P., Wesselink, W., Willemse, T.A.C.: An Overview of the mCRL2 Toolset and Its Recent Advances. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 199–213. Springer, Heidelberg (2013)
12. De Nicola, R., Vaandrager, F.: Three Logics for Branching Bisimulation. Journal of the ACM 42(2), 458–487 (1995)
13. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
14. van Glabbeek, R.J., Weijland, W.P.: Branching Time and Abstraction in Bisimulation Semantics. Journal of the ACM 43(3), 555–600 (1996)
15. Groote, J., Vaandrager, F.: An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 626–638. Springer, Heidelberg (1990)
16. Jeong, C., Kim, Y., Oh, Y., Kim, H.: A Faster Parallel Implementation of the Kanellakis-Smolka Algorithm for Bisimilarity Checking. In: ICS (1998)
17. Kanellakis, P., Smolka, S.: CCS Expressions, Finite State Processes, and Three Problems of Equivalence. In: PODC, pp. 228–240. ACM (1983)
18. Lee, I., Rajasekaran, S.: A Parallel Algorithm for Relational Coarsest Partition Problems and Its Implementation. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 404–414. Springer, Heidelberg (1994)
19. Orzan, S.: On Distributed Verification and Verified Distribution. Ph.D. thesis, Free University of Amsterdam (2004)
20. Paige, R., Tarjan, R.: A Linear Time Algorithm to Solve the Single Function Coarsest Partition Problem. In: Paredaens, J. (ed.) ICALP 1984. LNCS, vol. 172, pp. 371–379. Springer, Heidelberg (1984)
21. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
22. Wijs, A.J., Bošnački, D.: Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 98–116. Springer, Heidelberg (2012)
23. Wijs, A., Bošnački, D.: GPUexplore: Many-Core On-The-Fly State Space Exploration Using GPUs. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 233–247. Springer, Heidelberg (2014)

24. Wijs, A., Katoen, J.-P., Bošnački, D.: GPU-Based Graph Decomposition into Strongly Connected and Maximal End Components. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 310–326. Springer, Heidelberg (2014)
25. Wijs, A.J., Lisser, B.: Distributed Extended Beam Search for Quantitative Model Checking. In: Edelkamp, S., Lomuscio, A. (eds.) MoChArt IV. LNCS (LNAI), vol. 4428, pp. 166–184. Springer, Heidelberg (2007)
26. Wijs, A., van de Pol, J., Bortnik, E.: Solving Scheduling Problems by Untimed Model Checking - The Clinical Chemical Analyser Case Study. In: FMICS, pp. 54–61. ACM (2005)
27. Wu, Z., Liu, Y., Liang, Y., Sun, J.: GPU Accelerated Counterexample Generation in LTL Model Checking. In: Merz, S., Pang, J. (eds.) ICFEM 2014. LNCS, vol. 8829, pp. 413–429. Springer, Heidelberg (2014)
28. Zhang, S., Smolka, S.: Towards Efficient Parallelization of Equivalence Checking Algorithms. In: FORTE, North-Holland. IFIP Transactions, vol. C-10, pp. 121–135 (1992)