Int J Softw Tools Technol Transfer (2011) 13:21–35
DOI 10.1007/s10009-010-0176-4

SPIN 2009

Parallel probabilistic model checking on general purpose graphics processors

Dragan Bošnački · Stefan Edelkamp · Damian Sulewski · Anton Wijs

Published online: 13 October 2010

Abstract We present algorithms for parallel probabilistic model checking on general purpose graphic processing units (GPGPUs). Our improvements target the numerical components of the traditional sequential algorithms. In particular, we capitalize on the fact that in most of them operations like matrix–vector multiplication and solving systems of linear equations are the main complexity bottlenecks. Since linear algebraic operations can be implemented very efficiently on GPGPUs, the new parallel algorithms show considerable runtime improvements compared to their counterparts on standard architectures. We implemented our parallel algorithms on top of the probabilistic model checker PRISM. The prototype implementation was evaluated on several case studies in which we observed significant speedup over the standard CPU implementation of the tool.

Keywords Parallel model checking · General purpose graphics processing units

1 Introduction

1.1 Parallel model checking

Model checking is one of the most successful formal techniques for the verification of software and hardware systems. Developed at the beginning of the 1980s, nowadays it is used by major companies like Microsoft and Intel to improve the quality of their products. Although originally introduced as a fully automated technique for the verification of highly parallel and distributed systems, ironically enough, in the past model checking has mostly relied on sequential algorithms.

Indeed, parallel model checking algorithms for clusters of CPUs (e.g., [10,36,41]) or shared memory architectures [32,33] have been designed before. However, parallel implementations of model checking tools started having impact only recently. A notable example is the study on a big cluster (DAS-3) [4]. Other examples of cluster-oriented tools are the CADP toolbox and the LTSmin toolset. The construction and analysis of distributed processes (CADP) toolbox [24], used to design communication protocols and distributed systems also includes tools for distributed verification, e.g., the Distributor tool, which uses an algorithm based on [25] to perform exhaustive reachability analysis on a cluster of computers. LTSmin [13] is a toolset for manipulating labeled transition systems. The main tool LTSmin-mpi is a distributed implementation of signature-based bisimulation reduction for strong bisimulation and branching bisimulation. Lpo2lts-mpi is another tool from this toolset for distributed state space generation of mCRL2 models on cluster computers. The distributed and parallel verification environment provides also a parallel distributed-memory enumerative model-checking tool called DiVinE-Cluster [7] for the verification of concurrent systems. DiVinE Cluster can be used to prove correctness of verification models.

According to [37] less than two years ago a clear majority of the 500 most powerful computers in the world (http://www.top500.org) were characterized as clusters of computers/processors that work in parallel. Unfortunately, this has not had a major impact on the popularity of parallel computing both in industry and academia. With the emergence of the new parallel hardware technologies, like multi-core processors and general purpose graphics processing units, this

D. Bošnački (✉) · A. Wijs
Eindhoven University of Technology, Eindhoven, The Netherlands
e-mail: dragan@win.tue.nl

S. Edelkamp · D. Sulewski
TZI, Universität Bremen, Bremen, Germany

situation is changing drastically, which also reflects in the parallel algorithms for model checking.

In [30,31] the concept of multi-core model checking was introduced, followed by [6]. Several research groups either implemented multi-core parallelization into existing tools or developed a new checking software based on multi-core algorithms. Probably the most visible examples are the multi-core versions of SPIN [31] and DiVinE [9].

1.2 GPGPU programming

In recent years (general purpose) graphics processor units [(GP)GPUs] have become powerful massively parallel systems and they have outgrown their initial application niches in accelerating computer graphics. This has been facilitated by the introduction of new application programming interfaces (APIs) for general computation on GPUs, like the Compute Unified Device Architecture (CUDA) SDK form NVIDIA [18], Stream SDK from AMD [43], and Open CL [38]. Applications that exploit GPUs in different domains, like fluid dynamics, protein folding prediction in bioinformatics, Fast Fourier Transforms, and many others, have been developed in the last several years [39]. In model checking, however, GPUs were not used. To the best of our knowledge the first attempts to do model checking on GPUs were by the authors of this paper [14,20]. These efforts were followed by Barnat et al. [5]. Edelkamp et al. improved large-scale disk-based model checking by shifting complex numerical operations to the graphic card. As delayed elimination of duplicates is the performance bottleneck, the authors performed parallel processing on the GPU to improve the sorting speed significantly. Since existing GPU sorting solutions like Bitonic Sort and Quicksort do not obey any speedup on state vectors, they propose a refined GPU-based Bucket Sort algorithm. Additionally, they study sorting a compressed state vector and obtain speedups for delayed duplicate detection of up to one order of magnitude with a 8800-GTX GPU. Barnat et al. redesigned the maximal accepting predecessors algorithm for LTL model checking in term of matrix–vector product in order to accelerate LTL model checking on many-core GPU platforms. The drawback of this approach is the large amount of memory necessary on the graphics card. Since the whole state space is represented as a matrix only small problems can be analyzed.

1.3 Probabilistic model checking

Development of the probabilistic[1] model checking was instigated by the fact that probabilities are often an unavoidable ingredient of both natural and man-made systems. Therefore, in probabilistic model checking the satisfaction of properties is quantified with some probability. This is in contrast to traditional model checking which, at least in theory, tries to establish an absolute correctness or failure of the property. Probabilistic model checking has been proven to be a powerful framework for modeling various systems ranging from randomized algorithms via performance analysis to biological networks.

There is a significant overlap between the algorithms used in probabilistic model checking and their counterparts in traditional model checking. In both cases one the reachability of the underlying transition systems is computed. Still, there are also important differences because numerical methods are used to compute the transition probabilities. It is those numerical components that we are targeting in this paper and show how they can be sped up by employing the power of the new graphic processors technology.

1.4 Contribution

Traditionally the main bottleneck in practical applications of model checking has been the infamous state space explosion [44] and, as a direct consequence, large requirements in time and space. With the emergence of the new 64-bit processors there is no practical limit to the amount of shared memory that could be addressed. As a result the goals shift towards improving the runtime of the model checking algorithms [31]. In this paper we show that significant runtime gains can be achieved exploiting the power of GPUs in probabilistic model checking. This is because basic algorithms for probabilistic model checking are based on operations like solving linear equation systems and matrix–vector multiplication. These operations lend themselves to very efficient implementation on GPUs. Because of the massive parallelism—a standard commercial video card comprises hundreds of fragment processors—impressive speedups with regard to the sequential counterparts of the algorithms are quite common.

We present two algorithms that are parallel adaptations of the method of Jacobi for solving linear equations. Jacobi was chosen over other methods that usually outperform it on sequential platforms because of its lower memory requirements and potential to be parallelized because of fewer data dependencies. The algorithms feature sparse matrix–vector multiplication. It requires a minimal number of copy operations from RAM to GPU and back. We implemented the

[1] In the literature probabilistic and stochastic model checking often are used interchangeably. Usually a more clear distinction is made by relating the adjectives probabilistic and stochastic to the underlying model: discrete- and continuous-time Markov chain, respectively. For the sake of simplicity in this paper our focus is on discrete-time Markov chains, so we opted for consistently using the qualification "probabilistic". Nevertheless, as we also emphasize in the paper, the concepts and algorithms that we present here can be applied as well to continuous-time Markov chains.

Footnote 1 continued

algorithms on top of the probabilistic model checker PRISM [34]. The prototype implementations were evaluated on several case studies.

1.5 Related work

This paper extends a precursor work that was presented at SPIN '09 [14]. Besides editorial changes, e.g., clarity of the text, relevant related work, we implemented a 64-bit version of the GPU-based PRISM model checker and ran new experiments on one and on two graphics cards. For the latter setting, a synchronization mechanism for the communication had to be added. Besides this, to allow comparisons, we also implemented an alternative GPU-based Jacobi iteration method, using a backward, inclusive segmented scan provided by the CUDPP library [17].

In [12] a distributed model checking algorithm of Markov chains is presented. The approach in the paper focuses on continuous-time Markov chain models and Computational Stochastic Logic. They too use a parallel version of Jacobi's method, which is different from the one presented in this paper. This is reflected in the different memory management (GPUs hierarchical shared memory model versus the distributed memory model) and in the fact that their algorithm stores part of the state space on external memory (disks). Also, [12] is much more oriented towards increasing the state spaces of the stochastic models, than improving algorithm runtimes, which is our main goal. Maximizing the state space sizes of stochastic models by joining the storages of individual workstations of a cluster is the goal pursuit also in [16]. A significant part of the paper is on implicit representations of the state spaces with a conclusion that, although they can further increase the state space sizes, the runtime remains a bottleneck because of the lack of efficient solutions for the numerical operations.

In [1] a shared memory algorithm is introduced for CTMC construction and numerical steady-state solution constructing the CTMCs from generalized stochastic Petri nets. The algorithm for computing steady state probability distribution is an iterative one. Compared to this work, our algorithm is more general as it can be used in CTMCs also to compute transient probabilities.

Another shared memory approach is described in [8]. It targets Markov decision processes, which we do not consider in this paper. As such it differs from our work significantly since the quantitative numerical component of the algorithm reduces to solving systems of linear inequalities, i.e., using linear program solvers. In contrast, large-scale solutions support multiple scans over the search space on disks [19,21].

Enumerating state spaces in model checking shares similarities with solving AI search problems. In single-agent search challenges (e.g., sliding-tile puzzle, and instances to Peg-Solitaire, Frogs-and-Toads, Top-Spin, and Pancake) on the GPU Edelkamp et al. [22] established maximal speedups of up to factor 27 with regard to the single-core CPU computation. In all domains we arrive at one order of magnitude speedup. Based on the two-bit approach by Cooperman and Finkelstein [12] the authors proposed a general one-bit reachability and a specified one-bit BFS algorithm and showed how to speed up the search by parallelizing the algorithms and computing reversible minimal perfect hash functions on the GPU. Several perfect hash functions are studied, ranging from permutations to binomial coefficients. In *strongly* solving Nine-Men-Morris (for the first time) [23] the authors arrived at an overall speedup factor of over 12. For all three phases in the game, multinomial hashing for ranking and unranking states on the GPU, and a parallel retrograde analysis on the GPU were applied.

1.6 Layout

The rest of the paper is structured as follows. Section 2 briefly introduces probabilistic model checking. Section 3 describes the architecture, execution model and some challenges of GPU programming. Section 4 presents the algorithm for matrix–vector multiplication as used in each Jacobi iteration and its ports to the GPU. Section 5 evaluates our approach verifying examples shipped with the PRISM source showing significant speedups compared to the current CPU solution. The last section concludes the paper and discusses the results.

2 Probabilistic model checking

In this section we briefly recall the basics of probabilistic model checking along the lines of [35]. For simplicity reasons, we mainly focus on discrete-time Markov chains (DTMCs) and the logic PCTL, and only briefly discuss continuous-time Markov chains. The main point of this overview is to show that matrix vector multiplication and solving systems of linear equations are corner-stones of most of the algorithms for probabilistic model checking. More details can be found in, e.g., [2,35].

2.1 Discrete time Markov chains

Given a fixed finite set of atomic propositions AP we define a DTMC as follows:

Definition 1 A (labeled) DTMC $\mathcal{D}$ is a tuple $(S, \hat{s}, \mathbf{P}, L)$ where

- S is a finite set of states;
- $\hat{s} \in S$ is the initial state;
- $\mathbf{P} : S \times S \to [0, 1]$ is the transition probability matrix where $\Sigma_{s' \in S}\mathbf{P}(s, s') = 1$ for all $s \in S$;

- $L : S \to 2^{\mathrm{AP}}$ is a labeling function which assigns to each state $s \in S$ the set $L(s)$ of atomic propositions that are valid in the state.

Intuitively, real numbers $\mathbf{P}(s, s')$ in the interval $[0, 1]$ give the probability of a transition from s to s'. The definition implies that for each state the sum of the probabilities of the outgoing transitions must be 1. Consequently, end states, i.e., states which will normally not have outgoing transitions are modeled by adding self-loops with probability 1.

2.2 Probabilistic computational tree logic

Properties of DTMCs can be specified using *probabilistic computation tree logic* (PCTL) [26], which is a probabilistic extension of CTL.

Definition 2 PCTL has the following syntax:

$$\Phi ::= \mathtt{true} \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid P_{\sim p}[\phi]$$
$$\phi ::= \mathtt{X}\ \Phi \mid \Phi\ \mathtt{U}^{\leq k}\Phi$$

where $a \in AP$, $\sim \in \{<, \leq, \geq, >\}$, $p \in [0, 1]$, and $k \in \mathbb{N} \cup \{\infty\}$.

The above definition features both state formulae Φ and path formulae ϕ, which are interpreted on states and paths, respectively, of a given DTMC $\mathcal{D}$. However, the properties are specified exclusively as state formulae. Path formulae have only an auxiliary role and they occur as a parameter in state formulae of the form $P_{\sim p}[\phi]$. Intuitively, $P_{\sim p}[\phi]$ is satisfied in some state s of $\mathcal{D}$, if the probability of choosing a path that begins in s and satisfies ϕ is within the range given by $\sim p$. To formally define the satisfaction of the path formulae one defines a probability measure, which is beyond the scope of this paper. (For example, see [35] for more detailed information.) Informally, this measure captures the probability of taking a given finite path in the DTMC, which is calculated as the product of the probabilities of individual transitions of this path.

The path operators have intuitive meaning which is analogous to the one in standard temporal logics. The formula $\mathtt{X}\ \Phi$ is true if Φ is satisfied in the next state of the path. The bounded until formula $\Phi\ \mathtt{U}^{\leq k}\Psi$ is satisfied if Ψ is satisfied in one of the next k steps and Φ holds until this happens. For $k = \infty$ one obtains the unbounded until. In this case we omit the superscript and write $\Phi\ \mathtt{U}\ \Psi$. The interpretation of unbounded until is the standard strong until.

2.3 Algorithms for model checking PCTL

Given a labeled DTMC $\mathcal{D} = (S, \hat{s}, P, L)$ and a PCTL formula Φ, usually we are interested whether the initial state of $\mathcal{D}$, $\hat{s}$, satisfies Φ. Nevertheless, the algorithm works by checking the satisfaction of Φ for each state in S. The output of the algorithm is $Sat(\Phi)$, the set of all states that satisfy Φ.

The algorithm starts by first constructing the parse tree of the PCTL formula Φ. The root of the tree is labeled with Φ and each other node is labeled by a sub-formula of Φ. The leaves are labeled with `true` or an atomic proposition. Starting with the leaves, in a recursive bottom-up manner for each node n of the tree the set of states is computed that satisfies the sub-formula that labels n. When we arrive at the root we can determine $Sat(\Phi)$.

The model checking algorithms for the state PCTL formulae are analogous with their counterparts in CTL and as such quite straightforward to implement. The only exceptions are the path formulae whose algorithms contain an extensive numerical component that is used to compute the transition probabilities. They are the most computationally demanding part of the model checking algorithm and as such a logical target of our improvement via parallel algorithms for GPUs.

To get a general picture above these claims we consider the algorithm for the formulae of the form $\mathtt{P}[\Phi\ \mathtt{U}^{\leq k}\Psi]$, where $k = \infty$. The numerical component of this case reduces to finding the least solution of the linear equation system:

$$\mathbf{W}(s, \Phi\mathtt{U}\Psi) = \begin{cases} 1 & \text{if } s \in Sat(\Psi) \\ 0 & \text{if } s \in Sat(\neg\Phi \wedge \neg\Psi) \\ \Sigma_{s' \in S}\mathbf{P}(s, s') \cdot \mathbf{W}(s', \Phi\mathtt{U}\Psi) & \text{otherwise} \end{cases}$$

where $\mathbf{W}(\Phi\ \mathtt{U}\ \ \Psi)$ is the resulting vector of probabilities indexed by the states in S. Only the states in which the formula is satisfied with probabilities 1 and 0 have a special treatment. For each other state the probabilities are computed in a recurrent fashion using the corresponding probabilities of the neighboring states. Before solving the system, the algorithm employs some optimizations by precomputing the states that satisfy the formula with probability 0 or 1. The (simplified) system linear equations can be solved using iterative methods that comprise matrix–vector multiplication. One such method is presented by Jacobi, which is also one of the methods that PRISM uses and which we describe in more detail in Sect. 4. We choose Jacobi's method over methods that on sequential architectures usually perform better. This is because Jacobi has certain advantages in the parallel programming context. For instance, it has lower memory consumption compared to the Krylov subspace methods and less data dependencies than the Gauss–Seidel method, which makes it easier to parallelize [12]. The algorithms for the next operator and bounded until boil down to a single matrix–vector product and a sequence of such products, respectively.

PCTL can be extended with various rewards (costs) operators that we do not give here. The algorithms for those operators can also be reduced to matrix–vector multiplication [35].

Thus, the main runtime bottleneck of the probabilistic model checking algorithms is the computational part, and in particular the linear algebraic operations. Their share of the total runtime of the algorithms increases with the size of the model $|S|$. Model checking of a PCTL formula Φ on DTMC $\mathcal{D}$ is linear in $|\Phi|$, the size of the formula, and polynomial in $|S|$, the number of states of the DTMC. The most expensive are the operators for unbounded until and also the rewards operators which too boil down to solving system linear equations of size at most $|S|$. The complexity is linear in k_{max}, the maximal value of the bounds k in the bounded until formulae (which also occurs in some of the costs operators). However, usually this value is much smaller than $|S|$. So, for real world problems, that tend to have large state spaces, the dependence on the size $|S|$ is even more critical. In the sequel we show how by using parallel versions of the algorithms on GPU, one can obtain substantial speedups of more than one order of magnitude compared to the original sequential algorithms.

2.4 Beyond discrete time Markov chains

Matrix–vector product is also in the core of model checking continuous-time Markov chains, i.e., the corresponding computational stochastic logic (CSL) [3,12,35]. For instance, the next operator of CSL can be checked in the same way like its PCTL counterpart. Both algorithms for steady state and transient probabilities boil down to matrix–vector multiplication. On this operation hinge also various extensions of CSL with costs. Thus, the parallel version of the Jacobi algorithm that we present in the sequel, can be used also for stochastic models, i.e., models based on CTMCs.

3 GPU programming

The application of the modern GPUs goes far beyond the realm of graphics applications. They can be seen as general purpose multi-threaded data parallel co-processors. However, there are substantial architectural differences between GPUs and CPUs, including the new generations of multi-core processors. This imposes restrictions on the programs that can run on GPUs. Consequently, one has to cope with several new challenges when developing model checking algorithms for GPUs. The latter can significantly differ not only compared to their sequential counterparts, but also to the multi-core and distributed (cluster-based) analogues.

Harnessing the power of GPUs is facilitated by the new APIs for general computation on GPUs. CUDA is an interface from NVIDIA where programs are basically extended C programs. To this end CUDA features extensions like: special declarations to explicitly place variables in some of the memories (e.g., shared, global, local), predefined keywords (variables) containing the block and thread IDs, synchronization statements for cooperation between threads, runtime API for memory management (allocation, deallocation), and statements to launch functions on GPU.

In view of the above remarks, before describing our approach in more detail, we give an overview of the GPU architecture and the CUDA programming model [11].

3.1 CUDA programming model

CUDA enforces a program architecture which provides flexibility and minimizes the dependence of the software from the concrete GPU. A CUDA program consists of a *host* program which runs on the CPU and a set of CUDA *kernels*. The kernels, which are the parallel parts of the program, are launched on the GPU device from the host program, which comprises the sequential parts. The CUDA kernel is a parallel program that is executed as a set of *threads*. Each thread of the kernel executes the same code. Threads of a kernel are grouped in *blocks* that form a *grid*. Each thread block of the grid is uniquely identified by its block ID and analogously each thread is uniquely identified by its thread ID within its block. The dimensions of the thread and the thread block are specified at the time of launching the kernel. Blocks of a grid are ordered as a one- or two-dimensional array dividing the block ID in x, and y axis component, while the threads of a block are ordered in up to three dimensions. A thread is then identified uniquely by the x, y and z axis component of the thread ID.

3.2 CUDA memory model

In the CUDA memory model there are different kinds of memories that differ substantially in access speed (latencies). This has important implications for the efficiency of the CUDA programs.

The memory hierarchy loosely maps to the program thread-block-kernel hierarchy. Each thread has its own *on-chip registers* which are fast and *off-chip local memory*, which is quite slow. Per block there is a *shared memory*. Threads within a block cooperate via this memory which is on-chip and very fast. If more than one block is executed in parallel then the shared memory is equally split between them. The whole grid—all blocks and threads within them—have access to the off-chip *global memory*. The latter is large (up to 4 GB), but slow. The host has read and write access to the global memory (Video RAM, or VRAM), but cannot access the other memories (registers, local, shared). Thus, as such, global memory is used for communication between the host and the kernel. Threads within a block can communicate also via light-weight synchronization barriers.

3.3 GPU architecture and CUDA execution model

GPU architecture consists of a set of multiprocessors units called streaming multiprocessors (SMs). Each SM contains a set of processor cores called streaming processors (SPs). The NVIDIA GeForce GTX280, which we are using for the experiments in this paper, has 30 SMs each consisting of 8 SPs, which gives a total of 240 SPs.

Analogously with the memory model, there is a similar correspondence between the CUDA logical (programming) hierarchy and the physical (hardware) hierarchy of the GPU. Each thread is assigned to one processor (SP), whereas several threads can be executed on the same SP. Similarly, each block is mapped to one multiprocessor (SM), whereas each multiprocessor can execute several blocks. The logical kernel architecture allows flexibility: the GPU can schedule the blocks of the kernel depending on the concrete hardware architecture in an optimal way which is completely transparent for the user. Each multiprocessor performs computations in single instruction multiple threads (SIMT) manner, which means that each thread is executed independently with its own instruction address and local state (registers and local memory).

As already mentioned, threads are executed by the SPs and thread blocks are executed on the SMs. Each block is assigned to the same processor throughout the execution, i.e., it does not migrate. The number of blocks that can be physically executed in parallel on the same multiprocessor is limited by the number of registers and the amount of shared memory. Only one kernel at a time is executed per GPU.

3.4 GPU programming specifics

Due to the above described specific logical and physical architectures, GPU programs often require optimization techniques which are quite different compared to the multi-core and distributed parallel programming contexts. These idiosyncrasies of the GPU programming are mainly visible in the optimization of memory latencies, synchronization, thread mapping, the data layout in the memory, and data reuse.

Communication with the off-chip device memory is relatively slow compared to the enormous peak computational power. This is usually the main performance bottleneck. To fully exploit the capacity of the GPU parallelism this memory latency must be minimized. Another issue that can lead to a performance degradation is unnecessary synchronization between thread blocks. The inter-thread communication within a block is cheap via the fast shared memory, but the accesses to the global and local memories are more than hundred times slower.

Unlike the CPU threads, the GPU threads are very lightweight with negligible overhead of creation and switching. This allows GPUs to use thousands of threads whereas multi-core CPUs use only a few. Usually more threads and blocks are created than the number of SPs and SMs, respectively, which allows GPU to maximally use the capacity via smart scheduling—while some threads/blocks are waiting for data, the others which have their data ready are assigned for execution. Thus, another way to maximize the parallelism is by optimizing the thread mapping. This is often tightly coupled with the optimization of the memory access. One should strive towards an alignment of the data in the memory such that threads of the same block access memory locations which are as close as possible. In this case we have so-called coalesced accesses. Thus, threads that access physically close memory locations should be grouped together such that they can be provided data with the same memory access. Finally, in order to minimize the access to the slow global memory, one should exploit data reuse. The parts of the computation are localized to thread blocks which are synchronized as loosely as possible. These threads use local data as much as possible and the global results are written only at the end of the computation.

4 Matrix–vector multiplication and solving systems of linear equations on GPU

To speed up the algorithms we replace the sequential matrix–vector multiplication algorithm with a parallel one, which is adapted to run on GPU. In this section we describe our parallel algorithms which are derived from the Jacobi algorithm for solving linear equations. These algorithms were used for both bounded and unbounded until, i.e., also for solving systems of linear equations.

4.1 Jacobi iterations

As mentioned in Sect. 2 for model checking DTMCs, Jacobi iteration method is one option to solve the set of linear equations we have derived for until (U). Each iteration in the Jacobi algorithm involves a matrix–vector multiplication. Let n be the size of the state space, which determines the dimension $n \times n$ of the matrix to be iterated.

The formula of Jacobi for solving $Ax = b$ iteratively for an $n \times n$ matrix $A = (a_{ij})_{0 \le i,j \le n-1}$ and a current vector x^k is

$$x_i^{k+1} = 1/a_{ii} \cdot \left(b_i - \sum_{j \neq i} a_{ij} x_j^k \right), \quad \text{for } i \in \{0, \dots, n-1\}.$$

For better readability (and faster implementation), we may extract the diagonal elements and invert them prior to applying the formula. Setting $D_i = 1/a_{ii}$, $i \in \{0, \dots, n-1\}$ then yields

$$x_i^{k+1} = D_i \cdot \left(b_i - \sum_{j \neq i} a_{ij} x_j^k \right), \quad \text{for } i \in \{0, \dots, n-1\}.$$

The sufficient condition for Jacobi iteration to converge is that the magnitude of the largest eigenvalue (spectral radius) of matrix $D^{-1}(A - D)$ is bounded by value 1. Fortunately, the Perron–Frobenius theorem asserts that the largest eigenvalue of a (strictly positive) stochastic matrix is equal to 1 and all other eigenvalues are smaller than 1, so that $\lim_{k\to\infty} A^k$ exists. In the worst case, the number of iterations can be exponential in the size of the state space but in practice the number of iterations k until conversion to some sufficiently small ϵ according to a termination criteria, like $\max_i |x_i^k - x_i^{k+1}| < \epsilon$, is often moderate [42].

4.2 Sparse matrix representation

The size of the matrix is $\Theta(n^2)$, but for sparse models that usually appear in practice it can be compressed. Such matrix compaction is a standard technique used for probabilistic model checking and to this end special structures are used. In the algorithms that we present in the sequel we assume the so called *modified compressed sparse row/column format* [15]. We illustrate this format on the sparse transition probability matrix **P** given below:

row	0	0	1	1	2	2	2	3	4
col	2	4	2	3	0	3	4	0	0
non-zero	0.7	0.1	0.01	0.99	0.3	0.58	0.12	1	0.5

The above matrix contains only the non-zero elements of **P**. The arrays labeled *row*, *col*, and *non-zero* contain the row and column indices, and the values of the non-zero elements, respectively. More formally, for all r of the index range of the arrays, $\textit{non-zero}_r = \mathbf{P}(\textit{row}_r, \textit{col}_r)$. Obviously, this is already an optimized format compared to the standard full matrix representation. Still, one can save even more space as shown in the table below, which is, in fact, the above mentioned modified compressed sparse row/column format:

rsize	2	2	3	1	1				
col	2	4	2	3	0	3	4	0	0
non-zero	0.7	0.1	0.01	0.99	0.3	0.58	0.12	1	0.5

The difference with the previous representation is only in the top array *rsize*. Instead of the row indices, this array contains the row sizes, i.e., $\textit{rsize}_i$ contains the number of non-zero elements in row i of **P**. To extract row i of the original matrix **P**, we take the elements

$$\textit{non-zero}_{\textit{rstart}_i}, \dots, \textit{non-zero}_{\textit{rstart}_i + \textit{rsize}_i - 1}$$

where $\textit{rstart}_i = \sum_{k=0}^{i-1} \textit{rsize}_k$.

4.3 Algorithm implementation

The pseudo code of the sequential Jacobi algorithm that implements the aforementioned recurrent formula and which uses the compression given above is shown in Algorithm 1.

Algorithm 1 Jacobi iteration with row compression, as implemented in PRISM.

```
1: $k := 0$
2: $\textit{Terminate}$ := false
3: while (not $\textit{Terminate}$ and $k < \max_k$) do
4:    $h := 0$;
5:    for all $i := 0 \dots n$ do
6:       $d := b_i$;
7:       $l := h$;
8:       $h := l + \textit{rsize}_i - 1$;
9:       for all $j = l \dots h$ do
10:         $d = d - \left(\textit{non-zero}_j \cdot x^k_{\textit{col}_j}\right)$;
11:       $d := d \cdot D_i$;
12:       $x_i^{k+1} := d$;
13:    $\textit{Terminate}$ := true
14:    for all $i := 0 \dots n$ do
15:       if $|x_i^{k+1} - x_i^k| > \epsilon$ then
16:          $\textit{Terminate}$ := false
17:    $k := k + 1$;
```

The iterations are repeated until a satisfactory precision is achieved or the maximal number of iterations $\max_k$ is overstepped. In lines 6–8 (an element of) vector b is copied into the auxiliary variable d and the lower and upper bounds for the indices of the elements in array *non-zero* that correspond to row i are computed. In the for loop the product of row i and the result of the previous iteration, vector x^k, is computed. The new result is recorded in variable x^{k+1}.

Note that, since we are not interested in the intermediate results, only two vectors are needed: one, x, to store x^k, and another, x', that corresponds to x^{k+1}, the result of the current iteration. After each iteration the contents of x and x' are swapped, to reflect the status of x', which becomes the result of the previous iteration. We will use this observation to save space in the parallel implementation of the algorithm given below.

In lines 13–16 the termination condition is computed, i.e., it is checked if sufficient precision is achieved. We assume that vector x is initialized appropriately before the algorithm is started.

Due to the fact that the iterations have to be performed sequentially the matrix–vector multiplication is the part to be distributed. As a feature of the algorithm (that contributed most to the speedup) the comparison of the two solution vectors, x and x' in this case, is done in parallel. The GPU version of the Jacobi algorithm is given in Algorithms 2 and 3.

Algorithm 2, running on the CPU, copies vectors *non-zero* and *col* from the matrix representation, together with vectors

Algorithm 2 JacobiCPU: CPU part of the Jacobi iteration.

```
Require: $Devices$ = number of GPUs;
1: allocate memory for $x'$
2: allocate memory for col, non-zero, b, x, $\epsilon$, n and copy
3: allocate memory for TerminateGPU to be shared
4: $rstart_0 := 0$;
5: for $i = 1 \dots |rsize| + 1$ do
6:     $rstart_i := rstart_{i-1} + rsize_{i-1}$;
7: allocate memory for rstartGPU, copy rstart to rstartGPU
8: $k := 0$
9: for $d = 0 \dots Devices - 1$ do
10:    $Terminate_d$ := false
11: $blocks = (n/Devices)/BlockSize + 1$;
12: while (not $\bigwedge_{d=0}^{d<Devices} Terminate_d$ and $k < \max_k$) do
13:    for all $d = 0 \dots Devices - 1$ do in parallel
14:       if $Devices > 1$ then
15:          copy x to VRAM
16:       $r_{beg} = d * (n/Devices)$;
17:       $r_{end} = (d + 1) * (n/Devices)$;
18:       <<< $blocks$,BlockSize>>>JacobiKernel($r_{beg}, r_{end}$));
19:       copy TerminateGPU to $Terminate_d$;
20:       if $Devices > 1$ then
21:          copy x' to RAM
22:    Swap(x,x')
23:    $k = k + 1$;
24: copy x' to RAM;
```

x and b, and constants ϵ and n, to the global memory (VRAM) and allocates space for the vector x'. Having done this, space for the *Terminate* variable is allocated in the VRAM. Variable *rstart* defines the starting point of a row in the matrix array. The conversion from *rsize* to *rstart* is needed to let each thread find the starting point of a row immediately. (In fact, implicitly we use a new matrix representation where *rsize* is replaced with *rstart*.) Array *rstart* is copied to the global memory variable *rstartGPU*. To specify the number of blocks and the size of a block CUDA supports additional parameters in front of the kernel (<<< number of blocks, block size >>>). Here the grid is defined with $n/BlockSize + 1$ blocks,[2] and a fixed *BlockSize*. After the multiplication and comparison step on the GPU the *Terminate* variable is copied back and checked. This copy statement serves also as a synchronization barrier, since the CPU program waits until all the threads of the GPU kernel have terminated before copying the variable from the GPU global memory. If another iteration is needed x and x' are swapped.[3] After all iterations the result is copied back from global memory to RAM.

JacobiKernel shown in Algorithm 3 is the so-called kernel that operates on the GPU. Local variables d, l, h, i and j are located in the local registers and they are not shared between threads. The other variables reside in the global memory. The result is first computed in d (locally in each thread) and then written to the global memory (line 11). This approach

Algorithm 3 JacobiKernel: Jacobi iteration with row compression on the GPU.

```
Require: $r_{beg}$ beginning row for this kernel;
Require: $r_{end}$ ending row for this kernel;
1: $i := BlockId \cdot BlockSize + ThreadId$;
2: if ($i = 0$) then
3:    TerminateGPU := true;
4: $i := i + r_{beg}$;
5: if ($i < r_{end}$) then
6:    $d := b_i$;
7:    $l := rstartGPU_i$;
8:    $h := rstartGPU_{i+1} - 1$;
9:    for all $j = l \dots h$ do
10:      $d := d - non\text{-}zero_{j - rstartGPU_{r_{beg}}} \cdot x_{col_j}$;
11:   $d := d \cdot D_i$;
12:   $x'_i := d$;
13:   if $|x_i - x'_i| > \epsilon$ then
14:      TerminateGPU := false
```

minimizes the access to the global memory from threads. At invocation time each thread computes the row i of the matrix that it will handle. This is feasible because each thread knows its *ThreadId*, and the *BlockId* of its block. Note that the size of the block (*BlockSize*) is also available to each thread. Based on value i only one thread (the first one in the first block) sets the variable *TerminateGPU* to true. Recall, this variable is located in the global memory, and it is shared between all threads in all blocks. Now, each thread reads three values from the global memory (line 5 to 7), here we profit from coalescing done by the GPU memory controller. It is able to detect neighboring VRAM access and combine it. This means, if thread i accesses 2 bytes at b_i and thread $i + 1$ accesses 2 bytes at b_{i+1} the controller fetches 4 bytes at b_i and divides the data to serve each thread its chunk. In each iteration of the for loop an elementary multiplication is done. Due to the compressed matrix representation a double indirect access is needed here. As in the original algorithm the result is multiplied with the diagonal value D_i and stored in the new solution vector x'. Finally, each thread checks if another iteration is needed and consequently sets the variable *TerminateGPU* to FALSE. Concurrent writes are resolved by the GPU memory controller.

4.4 Algorithm using backward segmented scan

An alternative to the approach given in Sect. 4.3 for parallelizing the Jacobi iterations, is by using a so-called *segmented scan* algorithm, as e.g. provided by CUDPP, the CUDA Data Parallel Primitives Library [17]. In that case, each iteration includes several steps performed on the GPU. First, the product matrix is calculated, which is obtained by simply multiplying each element in the original matrix with the corresponding element in the vector. Second, a parallel segmented scan is performed, in which for each row, the sum of all the entries in the product matrix is determined. Finally,

2 If BlockSize is a divisor of n threads in the last block execute only the first line of the kernel.

3 Since C operates on pointers, only these are swapped in this step.

in a third parallel step, the entries representing the sums are extracted from the matrix, subtracted from the corresponding entries in the vector b, and multiplied with the corresponding entries in the inverted diagonal. The benefit of using such an approach is that many aspects of the calculation are highly parallelizable, both the multiplications and the additions. In contrast to algorithms proposed in e.g. [40], we integrate the parallel matrix–vector multiplication in an iterative method to solve linear equations. In that setting, the third parallel step needs to be different from the one used in the multiplication function provided by CUDPP; on top of extracting the right entries, our function completes the calculation of a Jacobi iteration.

We developed a parallel Jacobi algorithm for the GPU using the backward segmented scan function provided by CUDPP. This function computes the sums of all indicated parts of a given array in parallel, and places the results at the start of these parts. Hence, when providing the array representing a matrix, and indicating where each row starts by using a flags array, which can be derived in a parallel algorithm from the *rstart* array, the function computes all row sums.

Algorithm 4 describes the CPU part of the algorithm; first, an array f is constructed based on *rstart* as computed in Algorithm 2, the only difference being that for a row i, $f_i = -1$ if the row only consists of zero entries. Another notable difference from Algorithm 2 is the level of parallelism; the product matrix is computed in the *MultKernel* function (Algorithm 5), which can be performed on at most $|non\text{-}zero|$ threads instead of n threads. In Algorithm 5, furthermore, we allow each thread to compute the products for at most *EltsPerThread* different matrix entries.[4] The intuition behind this restriction is that we try to achieve that as much as possible SPs receive their data with one memory access. Once the product matrix is calculated, the row sums are computed in a *backward, inclusive* segmented scan. Technical details concerning this step can be found in e.g. [40]. We illustrate the function with an example. Consider the *prod* array displayed in the following table, and an accompanying *flags* array indicating the starting points of the rows with "T", i.e. the borders of the segments. In a backward, inclusive segmented scan, all elements in a segment are added, and the result is written at the segment starting point. In contrast to this, in an exclusive segmented scan, the final entry in a segment is not considered, and in a forward segmented scan, the result is written at the final position of a segment. A suitable *flags* array can be computed in a parallel algorithm, and in our case only needs to be computed once before the iterations are performed. In Algorithm 4, this is done in the function *setFlags*; all positions indicated by *rstart* are marked with "T" in the *flags* array.

[4] The CUDPP developers suggest *EltsPerThread* = 8, denoting the number of SPs in one MP.

prod	4	1	6	3	8	6	1	2	5
flags	T	F	F	T	F	T	F	F	F
result	11	7	6	11	8	14	8	7	5

Having obtained the result of the segmented scan, we complete the iteration with the *Gather* function (Algorithm 6), which is run on at most n threads; for each row i, the right entry from *prod* is extracted, if available, using the index provided by f_i. Of course, the sum equals 0 if row i only contains zero entries. We subtract the result from b_i, and multiply this with D_i. Also here, we allow each thread to process at most *EltsPerThread* entries, and *TerminateGPU* is updated after each entry computation to reflect the termination status.

Algorithm 4 JacobiCPU_SScan: The CPU part of the Jacobi iteration with backward segmented scan, for unbounded until computation.

```
1: allocate memory for $x'$
2: allocate memory for col, non-zero, b, x, $\epsilon$, n, prod, flags,
3: and copy
4: allocate memory for TerminateGPU to be shared
5: for $i = 0 \dots n-1$ do
6:   if $rstart_i = rstart_{i+1}$ then
7:     $f_i := -1$;
8:   else
9:     $f_i := rstart_i$;
10: allocate memory for fGPU, copy f to fGPU
11: $k := 0$
12: Terminate := false
13: $blocksRows = n/(BlockSize * EltsPerThread) + 1$;
14: $blocksElts = |non\text{-}zero|/(BlockSize * EltsPerThread) + 1$;
15: $<<< blocksRows, BlockSize >>>\ setFlags(rstart, flags)$;
16: while (not Terminate and $k < \max_k$) do
17:   $prod = <<< blocksElts, BlockSize >>>$MultKernel();
18:   $prod = <<< blocksRows, BlockSize >>>$
19:     cudppSegmentedScan($prod$);
20:   $<<< blocksRows,$ BlockSize$>>>$Gather($prod$);
21:   copy TerminateGPU to Terminate;
22:   Swap(x,x')
23:   $k = k + 1$;
24: copy $x'$ to RAM;
```

Algorithm 5 MultKernel: Computation of product matrix on the GPU.

```
1: $i := BlockId \cdot BlockSize + ThreadId$;
2: for all $j = 0 \dots EltsPerThread - 1$ do
3:   if ($i < |non\text{-}zero|$) then
4:     $prod_i := non\text{-}zero_i \cdot x_{col_i}$;
5:   $i := i + BlockSize$;
```

4.5 Multi-card exploration

In SLI or Tesla technologies more than one graphics card can be used for computation. We explain an extension of the

Algorithm 6 Gather: Extracting the final results for a Jacobi iteration on the GPU.

```
1: $i := BlockId \cdot BlockSize + ThreadId$;
2: $TerminateGPU :=$ **true**;
3: **for all** $j = 0 \ldots EltsPerThread - 1$ **do**
4:   **if** $(i < n)$ **then**
5:     $d := b_i$;
6:     **if** $fGPU_i \neq -1$ **then**
7:       $d := d - prod_{f_i}$;
8:     $d := d \cdot D_i$;
9:     $x'_i := d$;
10:    **if** $|x_i - x'_i| > \epsilon$ **then**
11:      $TerminateGPU :=$ **false**;
12:  $i := i + BlockSize$;
```

above setting to such a multi-card scenario. The core idea exemplified in a two-card setting is to split the matrix, in an upper and lower half. Here we gain global memory on the graphics card, as only half of the matrix has to be stored on one card. Then we multiply each part with the vector on a separate GPU. Each GPU has its own copy of the vector. Thus, each GPU computes one half of the vector—the first one the elements in the range $0, \ldots, N/2$ and the second one within the range $N/2, \ldots, N$. In each iteration of the outer loop (done by the CPU), i.e, after GPUs are done, we join the two halves together, update and copy the vector back to the GPU. The main difference between a single-card run and a multi-card scenario is the copying of the solution vector. While in a single-card scenario only the pointers to x and x' are swapped, in a multi-card scenario both cards have to be synchronized and each card have to receive the half of the solution computed by the other one. The termination condition remains a conjunct of the two termination conditions.

We may expect a slowdown that an extra copying of the vector could cause, but due to the increase of memory we expect larger problem instances to be validated. One additionally nice feature of the above scheme is that it scales seamlessly for an arbitrary number of GPUs.

5 Experiments

We conducted two sets of experiments, running on different machines to measure the impact of an additional graphics card. The cards are not identical but have the similar technical specifications. On each machine three protocols of different complexities were verified. In the second system an additional graphics card is available, so an empirical study of the multi-GPU algorithm is conducted. In the following three subsections we will describe the used hardware, the verified protocols and the achieved results in detail.

5.1 Used hardware

The experiments for the 32-bit version of our model checker were executed on (one core of) a personal computer with an AMD Athlon(tm) 64 X2 Dual-Core Processor 3800+ running at 2 GHz with 4 GB of RAM. This system includes a MSI N280GTX T20G graphic card with 1 GB global memory and 240 cores running at 0.6 GHz plugged into an PCI Express slot. The experiments for the 64-bit version of our model checker were executed on (one core of) a personal computer with Intel Core i7 CPU 920 running at 2.67 GHz providing 8 CPU cores. Here we used two NVIDIA geForce 285 GTX (MSI) graphics card with 1 GB VRAM and 240 streaming processors each running at 0.6 GHz. In this system RAM amounts to 12 GB.

The operating system in use on the 32 bit system was SUSE 11 with CUDA 2.1 SDK and the NVIDIA driver version 177.13. On the 64 bit system we use Ubuntu 9.04 with CUDA 2.2 SDK and the NVIDIA driver version 195.36.24.

The GTX200 chip on all cards contains 10 texture-processing clusters (TPCs). Each TPC consists of 3 streaming multiprocessors (SMs) and each SM includes 8 streaming processors (SPs) and 1 double-precision unit. In total, it has 240 SPs executing the threads in parallel. Maximum block size for this GPU is 512. Given a grid, the TPCs divide the blocks on its SMs, and each SM controls at most 1024 threads, which are run on the 8 SPs.

In all tables of this section n denotes the number of rows (columns) of the matrix, "iterations" denotes the number of iterations of the Jacobi method, "seq. time" and "par. time" denote the runtimes of the standard (sequential) version of PRISM and our parallel implementation extension of the tool, respectively. All times are given in seconds. The speedup is computed as the quotient between the sequential and parallel runtimes. Due to the different clock rates of the systems the theoretical speedup factor is $240 * 0.6$ GHz$/2$ GHz $= 72$ and $240 * 0.6$ GHz$/2.67$ GHz $= 54$ however due to memory latency on both sides such a factor is not to be expected in complex algorithms. All tables are partitioned into two parts, the first one showing the results for the 32-bit system, the second one for the 64-bit system. The first half does not contain the results for the multi-GPU implementation since a second GPU was not available in this system.

5.2 Verified protocols

We verified three protocols, `herman`, `cluster`, and `tandem`, shipped with the source of PRISM. The protocols were chosen due to their scalability and the possibility to verify properties whose checking boils down to linear algebraic operations. Different protocols show different speedups achieved by the GPU, because the Jacobi iterations are only

Table 1 Detailed information on the protocol properties

Protocol	Instance	n	Iterations	GPU mem (MB)
herman	15	32,768	245	55
herman	17	131,072	308	495
cluster	122	542,676	1,077	21
cluster	230	1,917,300	2,724	76
cluster	320	3,704,340	5,107	146
cluster	410	6,074,580	11,488	240
cluster	446	7,185,972	18,907	284
cluster	464	7,776,660	23,932	308
cluster	500	9,028,020	28,123	694
cluster	572	11,810,676	28,437	908
tandem	255	130,816	4,212	4
tandem	511	523,776	8,498	17
tandem	1,023	2,096,128	16,326	71
tandem	2,047	8,386,560	24,141	287
tandem	3,070	18,859,011	31,209	647
tandem	3,588	25,758,253	34,638	884
tandem	4,095	33,550,336	37,931	1,535

a part of the model checking algorithms, while the results show the time for the complete run.

The first protocol called `herman` is the Herman's self-stabilizing algorithm [28]. The protocol operates synchronously on an oriented ring topology, i.e., the communication is unidirectional. The instance number denotes the number of processes in the ring, which must be odd. The underlying model is a DTMC. We verified the PCTL property 3 from the property file `herman.pctl` (`R=? [F ''stable''{''k_tokens''} {max}]`). Table 1 identifies this protocol as the one with the fewest lines and iterations, but also reveals the matrix of being of the largest density showing that 131,072 lines take up 495 MB of the GPU memory.

The second case study is `cluster` [27] which models communication within a cluster of workstations. The system comprises two sub-clusters with N workstations in each of them, connected in a star topology. The switches connecting each sub-cluster are joined by a central backbone. All components can break down and there is a single repair unit to service all components. The underlying model is CTMC and the checked CSL property is property 1 from the corresponding property file (`S=? [ ''premium'' ]`). In this case study a sparser matrix was generated, which in turn needed more iterations to converge then the `herman` protocol. In the largest instance ($N = 572$) checked by the GPU, PRISM generates a matrix with 11,810,676 lines and iterates this matrix 28,437 times. It was even necessary to increase the maximum number of iterations, set by default to 10,000, to obtain a solution.

Table 2 Results for the `herman` protocol

Inst.	Seq. time	Par. time	Factor	2 GPUs	Factor
15	22.430	21.495	1.04		
17	304.108	206.174	1.48		
15	10.544	12.837	0.82	12.135	0.87
17	121.766	140.248	0.87	93.350	1.30

The third case study `tandem` is based on a simple tandem queueing network [29]. The model is represented as a CTMC which consists of a M/Cox(2)/1-queue sequentially composed with a M/M/1-queue. We use c to denote the capacity of the queues. We verified property 1 from the corresponding CSL property file (`R=? [ S ]`). For this protocol Table 1 denotes the largest sparsity allowing a matrix with 25,758,253 lines to occupy 884 MB of graphics card memory. Constant T was set to 1 for all experiments and parameter c was scaled as shown in Table 1.

5.3 Empirical results

Table 2 shows the results of the verification using our implementation of the algorithm described in Sect. 4.3. Even though the number of iterations is rather small compared to the other models, the GPU achieves a speedup factor of approx. 1.5, and 0.9 on the 64-bit system. Since everything beyond multiplication of the matrix and vector is done on the CPU, we have not expected a larger speedup. Unfortunately, it is not possible to scale up this model, due to the memory consumption being too high. The next possible instance (`herman19.pm`) consumes more then 2 GB. This table also reveals the differences between the used systems, while the clock time of the CPU differs only by about 30% the 64-bit system is more then twice as fast as the 32 bit one in the sequential mode. Here the GPU even slows down the verification process giving a factor of only 0.9. Due to the large density of the matrix adding a second GPU to the computation achieves a speedup in the larger instance, here the copying process, dominating the experiment, can be done in parallel to both GPUs, giving more time for the computation.

Figure 1 shows that GPU performs significantly better, Table 3 contains some exact numbers for chosen instances of the `cluster` protocol. The largest speedup reaches a factor of more then 9 on the 32-bit system and 6.6 on the other. Even for smaller instances, the GPU exceeds factor 2. In this protocol, as well as in the next one, for large matrices we observed a slight deterioration of the performance of the GPU implementation for which, for the time being, we could not find a clear explanation. One plausible hypothesis would be that after some threshold number of threads GPU cannot profit any more from smart scheduling to hide the memory

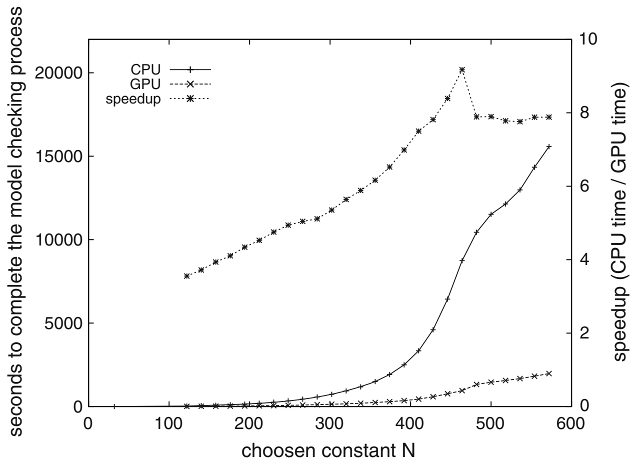

Fig. 1 Verification times for several instances of the `cluster` protocol on the 32-bit system. The x-axis shows the value of the parameter N. Speedup is computed as described in the text as a quotient between the runtime of standard PRISM and the runtime of our GPU extension of the tool

Table 3 Results for the `cluster` protocol. Parameter N is used to scale the protocol

N	Seq. time	Par. time	Factor	2 GPUs	Factor
122	31.469	8.855	3.55	.	
230	260.440	54.817	4.75	.	
320	931.515	165.179	5.63	.	
410	3,339.307	445.297	7.49	.	
446	6,440.959	767.835	8.38	.	
464	8,739.750	952.817	9.17	.	
500	11,516.716	1,458.609	7.89	.	
572	15,576.977	1,976.576	7.88	.	
122	16.400	6.906	2.37	8.620	1.90
230	135.269	34.732	3.89	46.685	2.90
320	469.827	101.664	4.62	141.456	3.32
410	1,649.663	286.014	5.77	429.626	3.84
446	3,143.487	512.708	6.13	785.629	4.00
464	4,270.262	643.850	6.63	1,024.335	4.17
500	4,865.687	1,027.095	4.73	1,470.256	3.30
572	6,630.097	1,386.101	4.78	1,964.418	3.38

The global memory usage (denoted as GPU mem) is in MB

latencies. This experiment shows the costs of synchronizing the graphics cards by the host. The speedup converges to about 4 and slows down the computation compared to the usage of one GPU.

In the `tandem` protocol the best speedup was recorded. For the best instance ($c = 2047$) PRISM generates a matrix with 8,386,560 rows, which is iterated 24,141 times. For this operation standard PRISM needs 9,672 s while our parallel implementation only needs 516 seconds, scoring a maximal speedup of a factor 18.7 on the 32 bit system. Even on the faster 64 bit system the speedup is over one order of magnitude for all larger instances. Here the effect of synchronizing is even more obvious. Using both GPUs a speedup of a factor larger than four seems not achievable despite the fact that larger instances can be checked (Table 4).

Table 4 Results from the verification of the `tandem` protocol

c	Seq. time	Par. time	Factor	2 GPUs	Factor
255	26.99	3.63	7.4	.	
511	190.26	17.80	10.7	.	
1,023	1,360.58	103.15	13.2	.	
2,047	9,672.19	516.33	18.7	.	
3,070	25,960.39	1,502.85	17.3	.	
3,588	33,820.21	2,435.41	13.9	.	
4,095	76,311.59	o.o.m		.	
255	14.96	3.56	4.20	6.79	2.20
511	98.81	11.42	8.65	27.89	3.54
1,023	658.78	65.51	10.06	166.27	3.97
2,047	3,642.62	384.68	9.47	946.76	3.85
3,070	10,049.93	866.27	11.60	2,510.67	4.00
3,588	15,114.93	1,319.13	11.46	3,794.79	3.98
4,095	22,174.01	o.o.m		5,386.44	4.12

The constant c is used to scale the protocol. Global memory usage, shown as GPU mem, is given in MB (o.o.m denotes out of global memory)

As mentioned above, 8 SPs share one double precision unit, but each SP has its own single precision unit. Hence, our hypothesis was that reducing the precision from double to single should bring a significant speedup. The code of PRISM was modified to support single precision floats for examining the effect. As can be seen in Fig. 2 the hypothesis was wrong. The time per iteration in double precision mode is nearly the same as the single precision mode. The graph clearly shows that the GPU is able to hide the latency which occurs when a thread is waiting for the double precision unit by letting the SPs work on other threads. Nevertheless, it is important to note that the GPU with single precision arithmetic was able to verify larger instances of the protocol, given that the floating point numbers consumed less memory.

It should be noted that in all case studies we also tried the MTBDD and hybrid representations of the models, which are an option in PRISM, but in all cases the runtimes were consistently slower than the ones with the sparse matrix representation, which are shown in the tables.

Table 5 presents the results obtained by verifying the protocols with an implementation of the GPU-based Jacobi algorithm with backward inclusive segmented scan, as described in Sect. 4.4. For this set of experiments the 64-bit system was used. We observe that speedups up to 15.1 are realized, and that it works faster than our other GPU-based Jacobi algorithm for the `herman` and `tandem` protocols. However, the algorithm tends to require more GPU memory.

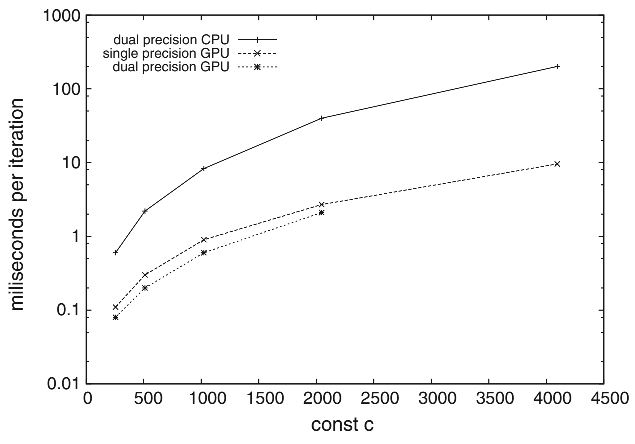

Fig. 2 Time per iteration on the 32-bit system in the `tandem` protocol. The CPU is significantly slower then the GPU operating in single or double precision. Reducing the precision has nearly no effect on the speed

Table 5 Results from the verification of the `herman`, `cluster`, and `tandem` protocol, respectively, using the GPU-based Jacobi algorithm with backward inclusive segmented scan

Protocol	Instance	Par. time	Factor	GPU mem. (MB)
herman	15	3.99	2.6	164
cluster	122	5.32	3.1	39
cluster	230	34.40	3.9	140
cluster	320	110.59	4.2	270
cluster	410	339.33	4.9	443
cluster	446	606.68	5.2	525
cluster	464	807.42	5.3	568
tandem	255	1.71	8.7	7
tandem	511	7.42	13.3	29
tandem	1,023	43.61	15.1	119
tandem	2,047	279.65	13.0	479

Global memory usage, shown as GPU mem, is given in MB, sequential times for factor computation given in tables above

The reason for this is that the segmented scan algorithm of CUDPP needs, in addition to the array of non-zero elements, a second array of equal size, filled with flags indicating whether a given position represents the start of a matrix row or not. In practice, this means that |*non-zero*| unsigned integers need to be stored, in addition to the requirements shared with the other GPU algorithm. The biggest gain in speed, compared to the other GPU algorithm, is obtained by the more rigorously parallelized construction of the product matrix, plus the employment of the segmented scan. However, for the `cluster` protocol, this does not pay off. The reason for this seems to be related to the absence of zeros in the intermediate vector results. In the `herman` 15 cases there are always some zeros present. In the `tandem` protocol cases, where the biggest gains can be observed, more than half the elements in the vectors are zeros. Finally, the intermediate results for the `cluster` protocol cases contain no zeros at all. Besides this, we could e.g. detect no fundamental differences (size, sparsity etc.) between the matrices of the different protocols, hence these cannot be the source of the different speedup results.

We believe that the presence (or absence) of zeros in the intermediate vectors has this effect on the computation time due to the way the segmented scan works. It seems that the scan algorithm can efficiently avoid computation when zeros are involved, while in the more naive approach of our other GPU algorithm, this can only be done to a lesser extent. On the other hand, if there are no zeros present, the scan turns out to be slower than simply adding up all the elements in a single row in the product matrix.

For immediate future work, we plan to investigate this more rigorously, and consider possibilities of reducing the memory requirements of the segmented scan method in the context of an iterative method such as Jacobi. For this, one possibility is to develop and implement a new segmented scan algorithm, based on existing ones.

6 Conclusions

In this paper we introduced GPU probabilistic/stochastic model checking as a novel concept. To this end we described two parallel versions of Jacobi's method for solving linear equations, which is the main core of the algorithms for model checking discrete- and continuous-time Markov chains, i.e., the corresponding logics PCTL and CSL. The algorithms were implemented on top of the probabilistic model checker PRISM. Their efficiency and the advantages of the GPU probabilistic model checking in general were illustrated on several case studies. Speedups of up to 18 times compared to the sequential implementation of PRISM were achieved. On a recent system the speedup still reaches a factor of 15.

We believe that our work opens a very promising research avenue on GPU model checking in general. To stay relevant for the industry, the area has to keep pace with the new technological trends. "Model checking for masses" gets tremendous opportunities because of the "parallelism for masses". To this end model checking algorithms that are designed for the verification of parallel systems and exploit the full power of the new parallel hardware will be needed.

In the future we intend to experiment with other matrix–vector algorithms, like:

– The Concurrent Number Cruncher library (CNC).[5] This provides a Jacobi-preconditioned Conjugate Gradient algorithm for matrix–vector multiplication.

[5] http://alice.loria.fr/index.php/software/4-library/34-concurrent-number-cruncher.html.

– Optimized sparse-matrix–vector multiplication for the GPU as proposed by IBM.[6]
– Adaption of the Gauss–Seidel algorithm.[7]

The Jacobi method does not converge for all linear equation systems. This can be resolved by introducing an under-relaxation parameter in the standard Jacobi. Furthermore, it may also be possible to accelerate the convergence of the standard Jacobi method by using an over-relaxation parameter. (The resulting method is known as Jacobi over-relaxation or JOR method).

Finally, a fine grained theoretical model of the GPU devices would be of very helpful in the development and analysis of GPU algorithms.

Acknowledgments We would like to thank the German Research Foundation (DFG) for the support in the project *Efficient Model Checking with GPUs and SSDs* (ED 74/8-1) and to the Netherlands Organisation for Scientific Research (NWO) for the support in the project *Efficient Multi-Core Model Checking* (pr. nr. 612.063.816).

References

1. Allmaier, S.C., Kowarschik, M., Horton, G.: State space construction and steady-state solution of GSPNs on a Shared-Memory Multiprocessor. In: Proceedings of 7th Intt. Workshop on Petri Nets and Performance Models (PNPM'97), pp. 112–121. IEEE Comp. Soc. Press (1997)
2. Baier, C., Katoen, J.-P.: Principles of Model Checking. pp. 950 MIT Press, Cambridge (2008)
3. Baier, C., Katoen, J.-P., Hermanns, H., Haverkort, B.: Model-checking algorithms for contiuous-time Markov chains. IEEE Trans. Softw. Eng. **29**(6), 524–541 (2003)
4. Bal, H., Barnat, J., Brim, L., Verstoep, K.: Efficient large-scale model checking. In: IEEE International Parallel & Distributed Processing Symposium (IPDPS) (2009)
5. Barnat, J., Brim, L., Ceska, M., Lamr, T.: CUDA accelerated LTL model checking. In: Proceedings of international conference on parallel and distributed systems (ICPADS), pp. 34–41 (2009)
6. Barnat, J., Brim, L., Ročkai, P.: Scalable multi-core model-checking. In: Model checking software, 14th international SPIN workshop, SPIN 07 LNCS, vol. 4595, pp. 187–203. Springer, Berlin (2007)
7. Barnat, J., Brim, L., Černá, I.: Cluster-based LTL model checking of large systems formal methods for components and objects. In: LNCS, vol. 4111, pp. 259–279. Springer, Berlin (2005)
8. Barnat, J., Brim, L., Černá, I., Ceska, M., Tumova, J.: ProbDiVinE-MC: multi-core LTL model checker for probabilistic systems. In: International conference on the quantitative evaluaiton of systems QEST 2008, pp. 77–78. IEEE Compuer Society Press (2008)
9. Barnat, J., Brim, L., Ročkai, P.: DiVinE multi-core—a parallel LTL model-checker. In: Automated technology for verification and analysis, ATVA 2008, LNCS, vol. 5311, pp. 234–239. Springer, Berlin (2008)
10. Barnat, J., Brim, L., Stríbrná, J.: Distributed LTL model checking in SPIN. In: Proceedings of the 8th Intl. Spin Workshop on Model Checking of Software, SPIN 2001, LNCS, vol. 2057, pp. 200–216. Springer, Berlin (2001)
11. Baskaran, M.M., Bordawekar, R.: Optimzing sparse matrix–vector multiplication on GPUs using compile-time and run-time strategies. IBM Reserach Report, RC24704 (W0812-047) (2008)
12. Bell, A., Haverkort, J.-P., Hermanns, B.R.: Distribute disk-based algorithms for model checking very large Markov chains. Formal Methods Syst. Design **29**, 177–196 (2006)
13. Blom, S., van de Pol, J., Weber, M.: LTSmin: Distributed and symbolic reachability. In: Proceedings of 22th Intl. Conf. Computer Aided Verification (CAV). LNCS, vol. 6174, pp. 354–359. Springer, Berlin (2010)
14. Bošnački, D., Edelkamp, S., Sulewski, D.: Efficient probabilistic model checking on general purpose graphics processors. In: Proceedings of 16th International SPIN Workshop, LNCS, vol. 3925, pp. 32–49. Springer, Berlin (2006)
15. Cooperman, G., Finkelstein, L.: New methods for using Cayley graphs in interconnection networks. Discrete Appl. Math. 37–38: 95118 (1992)
16. Ciardo, G.: Distributed and Structured Analysis Approaches to Study Large and Complex Systems. Eur. Educ. Forum School Formal Methods Perform. Anal. **2000**, 344–374 (2000)
17. CUDA Data Parallel Primitives Library. http://gpgpu.org/developer/cudpp
18. CUDA Programming Forum. http://www.nvidia.com/object/cuda_home.html
19. Dai, P., Mausam, Weld, D.S.: External memory value iteration. In: Proceedings of the Twenty-Third AAAI Conf. on Artificial Intelligence (AAAI), pp. 898–904 (2008)
20. Edelkamp, S., Sulewski, D.: Model Checking via Delayed Duplicate Detection on the GPU. Technical Report 821, Universität Dortmund, Fachberich Informatik, ISSN 0933-6192 (2008)
21. Edelkamp, S., Jabbar, S., Bonet, B.: External memory value iteration. In: Proceedigs of 17th Int. Conf. on Automated Planning and Scheduling, pp. 128–135, AAAI Press (2007)
22. Edelkamp, S., Sulewski, D., Yücel, C.: Perfect hashing for state space exploration on the GPU. In: Proceedings of 20th Int. Conf. International Conference on Automated Planning and Scheduling, pp. 57–64 (2010)
23. Edelkamp, S., Sulewski, D., Yücel, C.: GPU exploration of two-player games with perfect hash functions. In: Proceedings of third international symposium of combinatorial search (To appear)
24. Garavel, H., Mateescu, R., Lang, F., Serwe, W.: CADP 2006: A toolbox for the construction and analysis of distributed processes. In: Proceedings of 19th Intl. Conf. Computer Aided Verification (CAV), LNCS, vol. 4590, pp. 158–163. Springer, Berlin (2006)
25. Garavel, H., Mateescu, R., Smarandache, I.M.: Parallel State space construction for model-checking model checking software. In: 8th International SPIN Workshop, LNCS, vol. 2057, pp. 217–234. Springer, Berlin (2001)
26. Hansson, H., Jonsson, B.: A Logic for reasoning about time and reliability. Formal Aspects Comput. **6**(5), 512–535 (1994)
27. Haverkort, B., Hermanns, H., Katoen, J.-P.: On the use of model checking techniques for dependability evaluation. In: Proceedings of 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00), pp. 228–237 (2000)
28. Herman, T.: Probabilistic self-stabilization. Inform. Process. Lett. **35**(2), 63–67 (1990)
29. Hermanns, H., Meyer-Kayser, J., Siegle, M.: Multi terminal binary decision diagrams to represent and analyse continuous time Markov Chains. In: Proceedings of 3rd International Workshop on Numerical Solution of Markov Chains (NSMC'99), pp. 188–207 (1999)

[6] http://domino.watson.ibm.com/library/CyberDig.nsf/1e4115aea78b6e7c85256b360066f0d4/1d32f6d23b99f7898525752200618339?OpenDocument.

[7] http://portal.acm.org/citation.cfm?id=1581383.1582116.

30. Holzmann, G.J., Bošnački, D.: The Design of a multi-core extension of the Spin Model Checker. IEEE Trans. Softw. Eng. **33**(10), 659–674 (2007)
31. Holzmann, G.J., Bošnački, D.: Multi-core model checking with spin. In: Proceedings of Parallel and Distributed Processing Symposium, IPDPS 2007, pp. 1–8. IEEE International (2007)
32. Inggs, C.P., Barringer, H.: CTL* model checking on a shared memory architecture. Electronic Notes Theor. Comput. Sci. **128**(4), 107–123 (2005)
33. Inggs, C.P., Barringer, H.: Effective state exploration for model checking on a shared memory architecture. Electronic Notes Theor. Comput. Sci. **68**(4), 605–620 (2002)
34. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: Computer performance evaluation, modelling techniques and tools 12th international conference, TOOLS 2002, LNCS, vol. 2324, pp. 200–204. Springer, Berlin (2005)
35. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Formal methods for the design of computer, communication and software systems: performance evaluation, LNCS, vol. 4486, pp. 220–270. Springer, Berlin (2007)
36. Lerda, F., Sisto, R.: Distributed model checking in SPIN. In: Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, LNCS, vol. 1680, pp. 22–39. Springer, Berlin (1999)
37. Marowka, A.: Parallel computing on any desktop. Commun. ACM **50**(9), 75–78 (2007)
38. OpenCL: The open standard for parallel programming of heterogeneous systems http://www.khronos.org/opencl/
39. Philips, J.C., Braun, R., Wang, W., Gumbart, J., Tajkhorshid, E., Villa, E., Chipot, C., Skeel, R.D., Kale, L., Sculten, K.: Scalable molecular dynamics with NAMD. J. Comput. Chem. **26**, 1781–1802 (2005)
40. Sengupta, S., Harris, M., Zhang, Y., Owens, J.D.: Scan primitives for GPU computing. In: Graphics Hardware 2007, pp. 97–106 (2007)
41. Stern, U., Dill, D.: Parallelizing the Murϕ Verifier. In: Proceedings of 9th Intl. Conf. Computer Aided Verification (CAV), LNCS, vol. 1254, pp. 256–278. Springer, Berlin (1997)
42. Stewart, W.J.: Introduction to the Numerical Solution of Markov Chains. Princeton University Press, Princeton (1994)
43. ATI Stream technology http://www.amd.com/stream
44. Valmari, A.: The state explosion problem. In: Lectures on Petri Nets I: Basic Models, LNCS Tutorials, LNCS, vol. 1491, pp. 429–528. Springer, Berlin (1998)