Efficient Probabilistic Model Checking on General Purpose Graphics Processors

Dragan Bošnački[1], Stefan Edelkamp[2], and Damian Sulewski[3]

[1] Eindhoven University of Technology, The Netherlands
[2] TZI, Universität Bremen, Germany
[3] Technische Universität Dortmund, Germany

Abstract. We present algorithms for parallel probabilistic model checking on general purpose graphic processing units (GPGPUs). For this purpose we exploit the fact that some of the basic algorithms for probabilistic model checking rely on matrix vector multiplication. Since this kind of linear algebraic operations are implemented very efficiently on GPGPUs, the new parallel algorithms can achieve 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.

1 Introduction

Probabilistic Model Checking. Traditional model checking deals with the notion of absolute correctness or failure of a given property. On the other hand, probabilistic[1] model checking is motivated by the fact that probabilities are often an unavoidable ingredient of the systems we analyze. Therefore, the satisfaction of properties is quantified with some probability. This makes probabilistic model checking a powerful framework for modeling various systems ranging from randomized algorithms via performance analysis to biological networks.

From an algorithmic point of view, probabilistic model checking overlaps with the conventional one, since it too requires computing reachability of the underlying transition systems. 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] 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.

Parallel Model Checking. According to [31] less than two years ago a clear majority of the 500 most powerful computers in the world (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 situation is changing drastically.

This "parallelism for the masses" certainly offers great opportunities for model checking. Yet, ironically enough, model checking, that was mainly introduced for the verification of highly parallel systems, in the past has mostly relied on sequential algorithms. Parallel model checking algorithms have been designed before (e.g., [33, 30, 8]) and with few exceptions [27, 26] all of them targeted clusters of CPUs. However, this did not have any major impact in practice - besides some recent case studies on a big cluster (DAS-3)[4] none of the widely used model checking tools has a cluster version that preserves its full range of capabilities. In the last several years the things started to change. In [24, 25] the concept of multi-core model checking was introduced, followed by [5]. In the context of large-scale verification, different disk-based algorithms for solving the model checking problem have been published [16, 9, 7]. In [16], the authors avoid nested depth-first search for accepting cycle detection by reducing the liveness to a safety problem. This I/O-efficient solution was further improved by running directed search and exploiting parallelism. Another disk-based algorithm for LTL model checking [7] avoids the increase in space, but does not operate on-the-fly. The algorithm given in [9] is both on-the-fly and linear in the space requirements wrt. the size of the state space, but its worst-case time complexity is large. Other advances in large-scale LTL model checking exploit Flash media [18, 19].

GPGPU Programming. In the 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 CUDA form NVIDIA, Stream SDK from AMD, and Open CL. 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 [32]. In model checking, however, GPUs have not had any impact. To the best of our knowledge the only attempt to use model checking on GPUs was by the authors of this paper [15]. They 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 speed-up on state vectors, they propose a refined GPU-based Bucket Sort algorithm. Additionally, they study sorting a compressed state vector and obtain speed-ups for delayed duplicate detection of more than one order of magnitude with a 8800-GTX GPU.

Contribution. Traditionally the main bottleneck in practical applications of model checking has been the infamous state space explosion [35] 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 [25]. 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 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 – quite impressive speedups with regard to the sequential counterparts of the algorithms are quite common.

We present an algorithm that is a parallel adaptation of the method of Jacobi for matrix-vector product. 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 algorithm features sparse matrix vector multiplication. It requires a minimal number of copy operations from RAM to GPU and back. We implemented the algorithm on top of the probabilistic model checker PRISM [28]. The prototype implementation was evaluated on several case studies and remarkable speedups (up to factor 18) were achieved compared to the sequential version of the tool.

Related Work. In [11] a distributed algorithm for model checking of Markov chains is presented. 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 vs. the distributed memory model) and in the fact that their algorithm stores part of the state space on external memory (disks). Also, [11] 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 [12]. 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. The CTMCs are constructed 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 [6]. 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 [17, 13].

Layout. 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 the Jacobi iteration method and its port 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 along the lines of [29] the basics of probabilistic model checking for discrete-time Markov chains (DTMCs). More details can be found in, e.g., [29, 2].

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;*
- *$\mathbf{P} : S \times S \rightarrow [0, 1]$ is the transition probability matrix where $\Sigma_{s' \in S} \mathbf{P}(s, s') = 1$ for all $s \in S$;*
- *$L : S \rightarrow 2^{AP}$ 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.*

Each $\mathbf{P}(s, s')$ gives the probability of a transition from s to s'. For each state the sum of the probabilities of the outgoing transitions must be 1. Thus, end states, i.e., states which will normally not have outgoing transitions are modeled by adding self-loops with probability 1.

Probabilistic Computational Tree Logic. Properties of DTMCs can be specified using *Probabilistic Computation Tree Logic (PCTL)* [20], which is a probabilistic extension of CTL.

Definition 2. *PCTL has the following syntax:*

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

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

For the sake of presentation, in the above definition we use 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 [29] 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 intuitive meaning of the path operators is analogous to the ones 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 one.

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 $\mathcal{D}$ 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 subformula 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 subformula that labels n. When we arrive at the root we can determine $Sat(\Phi)$.

Except for the path formulae the model checking of PCTL formulae is actually the same as for their counterparts in CTL and as such quite straightforward to implement. In this paper we concentrate on the path formulae. They are the most computationally demanding part of the model checking algorithm and as such they are the targets of our improvement via GPU algorithms.

To give a general flavor of the path formulae, we give a briefly consider the algorithm for the formulae of the form $\mathtt{P}[\Phi\ \mathtt{U}^{\leq k}\Psi]$, where $k = \infty$. This case boils down to finding the least solution of the linear equation system:

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

where $\mathbf{W}(\Phi\ \mathtt{U}\quad \Psi)$ is the resulting vector of probabilities indexed by the states in S. The states in which the formula is satisfied with probabilities 1 and 0 are singled out. For each other state the probabilities are computed via 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 the one by Jacobi, which is also one of the methods that PRISM uses and which we describe in more detail in Section 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 ti easier to parallelize [11].

The algorithms for the next operator and bounded until boil down to a single matrix-vector product and a sequence of such products, respectively. Therefore they can also be resolved by using Jacobi's method.

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 [29].

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 also 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 $|S|$. Thus, the main runtime bottleneck of the probabilistic model checking algorithms remain the linear algebraic operations. Their share of the total runtime of the algorithms increases with $|S|$. So, for real world problems, that tend to have large state spaces, this dependence 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.
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) [29, 3, 11]. For instance, the next operator of CSL can be checked with 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

A considerable part of the challenges that arise in model checking algorithms for GPUs is due to the specific architectural differences between GPUs and CPUs and the restrictions on the programs that can run on GPUs. Therefore, before describing our approach in more detail, we give an overview of the GPU architecture and the Compute Unified Device Architecture (CUDA) programming model by the manufacturer NVIDIA [14] along the lines of [10].

Modern GPUs are no longer dedicated only to graphics applications. Instead a GPU can be seen as a general purpose multi-threaded massively data parallel co-processor. Harnessing the power of GPUs is facilitated by the new APIs for general computation on GPUs.

CUDA is an interface by NVIDIA which is used to program GPUs. CUDA 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.

CUDA Programming Model. 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 kernel that is executed on a set of threads. Each thread of the kernel executes the same code. Threads of a kernel are grouped in blocks. 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. The grid can be one- or two-dimensional and the blocks are at most three-dimensional..

CUDA Memory Model. Threads have access to different kind of memories. Each thread has its own on-chip registers and off-cheap local memory, which is quite slow. Threads within a block cooperate via shared memory which is on-cheap and very fast. If more than one block are executed in parallel then the shared memory is equally split between them. All blocks have access to the device memory which is large (up to 4GB), but slow since, like the local memory, it is not cached. 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. Besides the memory communication, threads within a block can cooperate via light-weight synchronization barriers.

GPU Architecture. The architecture of GPU features a set of multiprocessors units called streaming multiprocessors (SMs). Each of those contains a set of processor cores called streaming processors (SPs). The NVIDIA GeForce GTX280 has 30 SMs each consisting of 8 SPs, which gives in total 240 SPs.

CUDA Excution Model. Each block is mapped to one multiprocessor whereas each multiprocessor can execute several blocks. The logical kernel architecture allows flexibility to the GPU to schedule the blocks of the kernel depending of the concrete hardware architecture in an optimal and for the user completely transparent way. Each multiprocessor performs computations in SIMT (Single Instruction Multiple Threads) manner, which means that each thread is executed independently with its own instruction address and local state (registers and local memory). 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.

GPU Programming Challenges. To fully exploit the computational power of the GPUs some significant challenges will have to be addressed.

The main performance bottleneck is usually the relatively slow communication (compared to the enormous peak computational power) with the off-chip device memory. To fully exploit the capacity of the GPU parallelism this memory latency must be minimized. Further, it is recommended to avoid 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.

Another way to maximize the parallelism is by optimizing the thread mapping. Unlike the CPU threads, the GPU threads are very light-weight 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. Thread mapping is coupled with the memory optimization in the sense that threads that access physically close memory locations should be grouped together.

4 Matrix-Vector Multiplication 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 matrix-vector multiplication. This algorithm was used for both bounded and unbounded until, i.e., also for solving systems of linear equations.

Jacobi Iterations. As mentioned in Section 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 $A = (a_{ij})_{0 \leq i,j \leq n-1}$ and a current vector x^k is

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

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

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

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 iteration 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 [34].

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* [11]. We illustrate this format on the sparse transition probability matrix $\mathbf{P}$ given below:

row	0	0	0	1	1	2	2	2	3	4	4
col	1	2	4	2	3	0	3	4	0	0	2
non-zero	0.2	0.7	0.1	0.01	0.99	0.3	0.58	0.12	1.0	0.5	0.5

The above matrix contains only the non-zero elements of $\mathbf{P}$. 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	3	2	3	1	2						
col	1	2	4	2	3	0	3	4	0	0	2
non-zero	0.2	0.7	0.1	0.01	0.99	0.3	0.58	0.12	1.0	0.5	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 $\mathbf{P}$. To extract row i of the original matrix $\mathbf{P}$, we take the elements

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

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

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

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

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.

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 x and b, and constants ϵ and n, to the global memory (VRAM) and allocates space for the vector x'. Having

Algorithm 2 JacobiCPU: CPU part of the Jacobi iteration, for unbounded until computation.

```
1: allocate global memory for x'
2: allocate global memory for col, non-zero, b, x, $\epsilon$, n and copy them
3: allocate global memory for TerminateGPU to be shared between blocks
4: $rstart_0 := 0$;
5: for $i = 1 \ldots |rsize| + 1$ do
6:     $rstart_i := rstart_{i-1} + rsize_{i-1}$;
7: allocate global memory for rstartGPU and copy rstart to rstartGPU
8: $k := 0$
9: $Terminate := false$
10: while (not Terminate and $k < \max_k$) do
11:     <<<n/BlockSize+1,BlockSize>>>JacobiKernel();
12:     copy TerminateGPU to Terminate;
13:     Swap(x,x')
14:     $k = k + 1$;
15: copy x' to RAM;
```

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, implicitely we use a new matrix representatin 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 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,

[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.

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

```
1: $i := BlockId \cdot BlockSize + ThreadId$;
2: if $(i = 0)$ then
3:     $TerminateGPU := true$;
4: if $(i < n)$ then
5:     $d := b_i$;
6:     $l := rstartGPU_i$;
7:     $h := rstartGPU_{i+1} - 1$;
8:     for all $j = l \ldots h$ do
9:         $d := d - \textit{non-zero}_j \cdot x_{col_j}$;
10:    $d := d \cdot D_i$;
11:    $x'_i := d$;
12:    if $|x_i - x'_i| > \epsilon$ then
13:        $TerminateGPU := false$
```

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.

The implementation in Algorithm 2 matches the one for bounded-until ($\mathtt{U}^{\leq \mathtt{k}}$), except that bounded-until has a fixed upper bound on the number of iterations, while for until a termination criterion applies.

5 Experiments

All experiments were done on a PC with an AMD Athlon(tm) 64 X2 Dual Core Processor 3800+ with 4 GB of RAM; the operating system is 64-bit SUSE 11 with CUDA 2.1 SDK and the NVIDIA driver version 177.13. This system includes a MSI N280GTX T20G graphic card with 1 GB global memory plugged into an ExpressPCI slot. The GTX200 chip on this card contains 10 texture processing clusters (TPC). Each TPC consists of 3 streaming multiprocessors (SM) 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.

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 its properties by solving a linear function with the Jacobi method. Different protocols show different speedups achieved by the GPU, because the Jacobi iterations are only a part of the model checking algorithms, while the results show the time for the complete run.

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.

The first protocol called `herman` is the Herman's self-stabilizing algorithm [22]. The protocol operates synchronously on an oriented ring topology, i.e., the communication is unidirectional. The number in the file name 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 shows the results of the verification. Even though the number of iterations is rather small compared to the other models, the GPU achieves a speedup factor of approx. 1.5. 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 1 GB.

Table 1. Results for the `herman` protocol.

instance	n	iterations	seq. time	par. time	speedup
herman15.pm	32,768	245	22.430	21.495	1.04
herman17.pm	131,072	308	304.108	206.174	1.48

The second case study is `cluster` [21] 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" ]`). Fig. 1 shows that GPU performs significantly better, Table 2 contains some exact numbers for chosen instances. The largest speedup reaches a factor of more then 9. Even for smaller instances, the GPU exceeds factor 3. 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. 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 latencies.

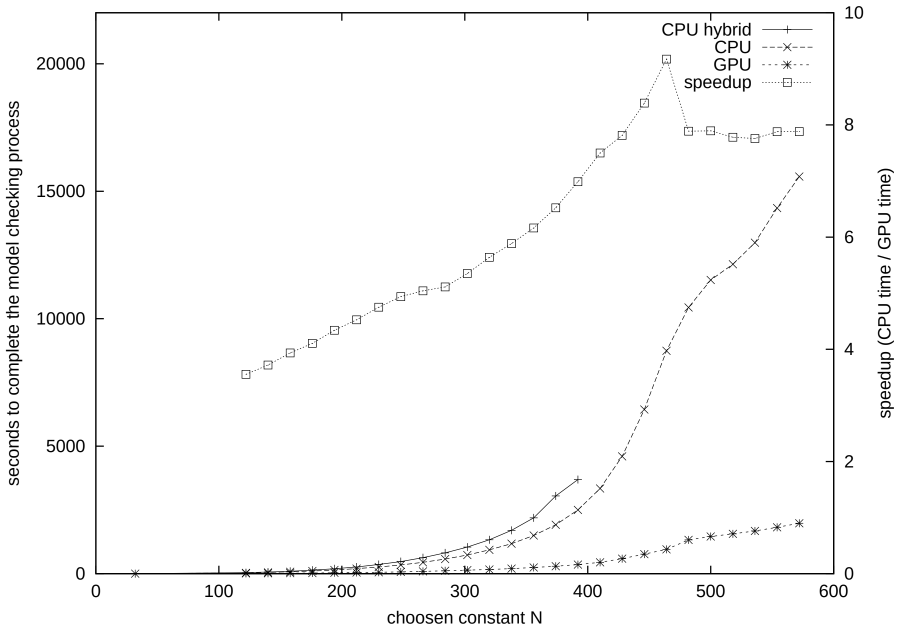

Fig. 1. Verification times for several instances of the `cluster` protocol. The x-axis shows the value of the parameter N. Speedup is computed as described in the text as a quotient between the runtime of standard PRISM and the runtime of our GPU extension of the tool.

The third case study `tandem` is based on a simple tandem queueing network [HMKS99]. 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 ]`). Constant T was set to 1 for all experiments and parameter c was scaled as shown in Table 3. In this 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 $9,672$ seconds while our parallel implementation only needs 516 seconds, scoring a maximal speedup of a factor 18.7.

As mentioned above, 8 SPs share one double precision unit, but each SP has an own single precision unit. Hence, our hypothesis was that reducing the precision from single to double should bring a significant speedup. The code of

Table 2. Results for the `cluster` protocol. Parameter N is used to scale the protocol. The global memory usage (denoted as GPU mem) is in MB.

N	n	iterations	seq. time	par. time	GPU mem	speedup
122	542,676	1,077	31.469	8.855	21	3.55
230	1,917,300	2,724	260.440	54.817	76	4.75
320	3,704,340	5,107	931.515	165.179	146	5.63
410	6,074,580	11,488	3,339.307	445.297	240	7.49
446	7,185,972	18,907	6,440.959	767.835	284	8.38
464	7,776,660	23,932	8,739.750	952.817	308	9.17
500	9,028,020	28,123	11,516.716	1,458.609	694	7.89
572	11,810,676	28,437	15,576.977	1,976.576	908	7.88

Table 3. Results from the verification of the `tandem` protocol. 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)

c	n	iterations	seq. time	par. time	GPU mem	speedup
255	130,816	4,212	26.994	3.639	4	7.4
511	523,776	8,498	190.266	17.807	17	10.7
1,023	2,096,128	16,326	1,360.588	103.154	71	13.2
2,047	8,386,560	24,141	9,672.194	516.334	287	18.7
3,070	18,859,011	31,209	25,960.397	1,502.856	647	17.3
3,588	25,758,253	34,638	33,820.212	2,435.415	884	13.9
4,095	33,550,336	37,931	76,311.598	o.o.m		

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 cases 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.

6 Conclusions

In this paper we introduced GPU probabilistic/stochastic model checking as a novel concept. To this end we described a parallel version of Jacobi's method for sparse matrix-vector multiplication, which is the main core of the algorithms

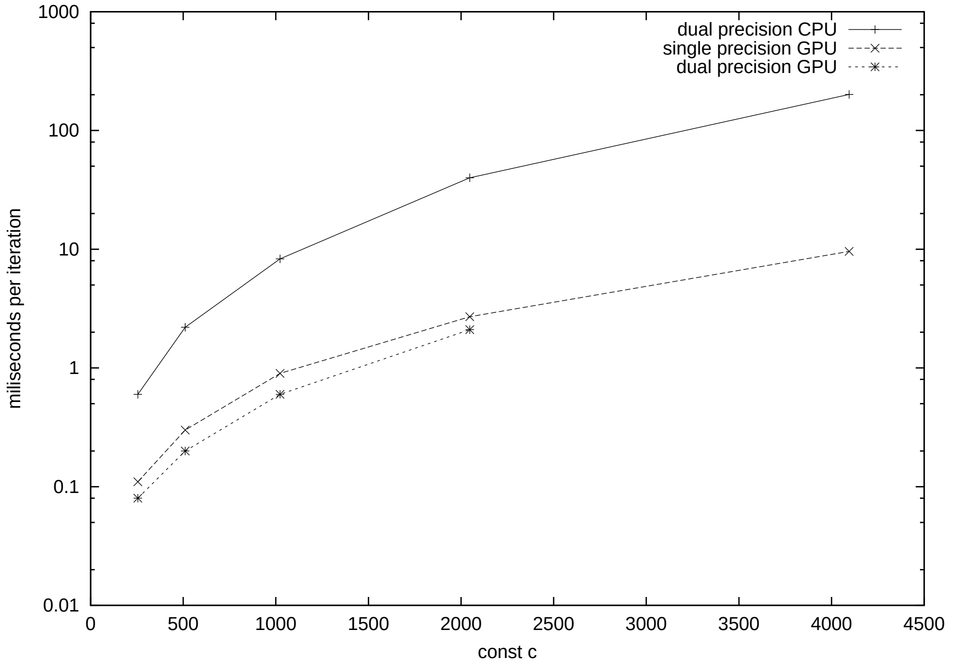

Fig. 2. Time per iteration in the `tandem` protocol. The CPU is significantly slower then the GPU operating in single or double precision. Reducing the precision has nearly no effect on the speed.

for model checking discrete- and continuous-time Markov chains, i.e., the corresponding logics PCTL and CSL. The algorithm was implemented on top of the probabilistic model checker PRISM. Its 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.

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 for GPUs, as well as with combination of multi-core and/or multi-GPU systems. What is needed for analyzing the time complexity of GPU algorithms is a fine grained theoretical model of its operation.

References

1. S.C. Allmaier, M. Kowarschik, G. Horton, *State Space Construction and Steady-state Solution of GSPNs on a Shared-Memory Multiprocessor*, Proc. 7th Intt. Workshop on Petri Nets and Peformance Models (PNPM'97), pp. 112-121, IEEE Comp. Soc. Press, 1997.
2. C. Baier, J.-P. Katoen, *Principles of Model Checking*, MIT Press, 950 pp, 2008.
3. C. Baier, J.-P. Katoen, H. Hermanns, B. Haverkort, *Model-Checking Algorithms for Contiuous-Time Markov Chains*, IEEE Transactions on Software Engineering, 29(6):524-541, 2003.
4. H. Bal, J. Barnat, L. Brim, and K. Verstoep, *Efficient Large-Scale Model Checking.* IEEE International Parallel & Distributed Processing Symposium (IPDPS). To appear, 2009.
5. J. Barnat, L. Brim, P. Ročkai, *Scalable Multi-core Model-Checking*, Model Checking Software, 14th International SPIN Workshop, SPIN 07 LNCS 4595, pp. 187-203, Springer, 2007.
6. J. Barnat, L. Brim, I. Černá, M. Ceska, J. Tumova, *ProbDiVinE-MC: Multi-core LTL Model Checker for Probabilistic Systems*, International Conference on the Quantitative Evaluaiton of Systems QEST 2008, pp. 77-78, IEEE Compuer Society Press, 2008.
7. J. Barnat, L. Brim, and P. Šimeček. I/O efficient accepting cycle detection. In *CAV*, volume 4590 of *LNCS*, pages 281–293. Springer, 2007.
8. J. Barnat, L. Brim, J. Stríbrná, *Distributed LTL Model Checking in SPIN*, Proc. of the 8th Intl. Spin Workshop on Model Checking of Software, SPIN 2001, LNCS 2057, pp. 200-216, Springer, 2001.
9. J. Barnat, L. Brim, P. Šimeček, and M. Weber. Revisiting resistance speeds up I/O-efficient LTL model checking. In *TACAS*, pages 48–62, 2008.
10. M.M. Baskaran, R. Bordawekar, *Optimzing Sparse Matrix-Vector Multiplication on GPUs Using Compile-time and Run-time Strategies*, IBM Reserach Report, RC24704 (W0812-047), 2008.
11. A. Bell, B.R. Haverkort, *Distribute Disk-based Algorithms for Model Checking Very Large Markov Chains*, Formal Methods in System Design 29:177-196, Springer, 2006.
12. G. Ciardo, *Distributed and Structured Analysis Approaches to Study Large and Complex Systems*, European Educational Forum: School on Formal Methods and Performance Analysis 2000: 344-374, 2000.
13. P. Dai, Mausam and D. S. Weld. *External Memory Value Iteration.* In Proc. of the Twenty-Third AAAI Conf. on Artificial Intelligence (AAAI), pages 898-904, 2008.
14. `http://www.nvidia.com/object/cuda_home.html#`
15. S. Edelkamp, D. Sulewski, *Model Checking via Delayed Duplicate Detection on the GPU*, Technical Report 821, Universität Dortmund, Fachberich Informatik, ISSN 0933-6192, 2008.
16. S. Edelkamp and S. Jabbar. *Large-scale directed model checking LTL.* In Proc. 13th International SPIN Workshop, LNCS 3925, pp. 1–18, Springer, 2006.
17. S. Edelkamp and S. Jabbar and B. Bonet. *External Memory Value Iteration.* In Proc. 17th Int. Conf. on Automated Planning and Scheduling, pp. 128–135, AAAI Press, 2007.
18. S. Edelkamp, P. Sanders, and P. Šimeček. Semi-external LTL model checking. In Proc. 20th Int. Conf. Computer Aided Verification, LNCS 5123, pp. 530–542, Springer, 2008.

19. S. Edelkamp and D. Sulewski. Flash-efficient LTL model checking with minimal counterexamples. In Software Engineering and Formal Methods, 73–82, 2008.
20. H. Hansson, B. Jonsson, *A Logic for reasoning about time and reliability*, Formal Aspects of Computing, 6(5):512-535, 1994.
21. B. Haverkort, and H. Hermanns, J.-P. Katoen, *On the Use of Model Checking Techniques for Dependability Evaluation*, Proc. 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00), pp. 228-237, 2000.
22. T. Herman, *Probabilistic Self-stabilization*, Information Processing Letters, **35** (2), pp. 63-67, 1990.
23. H. Hermanns, J. Meyer-Kayser, M. Siegle, *Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains*, Proc. 3rd International Workshop on Numerical Solution of Markov Chains (NSMC'99), pp. 188-207, 1999.
24. G.J. Holzmann, D. Bošnački, *The Design of a multi-core extension of the Spin Model Checker* IEEE Trans. on Software Engineering, **33** (10), pp. 659-674, October 2007. (first presented at: Formal Methods in Computer Aided Design (FMCAD), San Jose, November 2006.)
25. G.J. Holzmann, D. Bošnački, *Multi-core Model Checking with Spin,* Proc. Parallel and Distributed Processing Symposium, IPDPS 2007, pp. 1-8, IEEE International, 2007.
26. C.P. Inggs, H. Barringer, *CTL* Model Checking on a Shared Memory Architecture*, Electronic Notes in Theoretical Computer Science, **128** (4), pp. 107-123, 2005.
27. C.P. Inggs, H. Barringer, *Effective State Exploration for Model Checking on a Shared Memory Architecture*, Electronic Notes in Theoretical Computer Science, **68** (4), 2002.
28. M.Z. Kwiatkowska, G. Norman, D. Parker, *PRISM: Probabilistic Symbolic Model Checker*, Computer Performance Evaluation, Modelling Techniques and Tools 12th International Conference, TOOLS 2002, LNCS 2324, pp.200-204, Springer, 2005.
29. M. Kwiatkowska, G. Norman, D. Parker. *Stochastic Model Checking*, Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation, LNCS 4486, pp. 220-270, Springer, 2007.
30. F. Lerda, R. Sisto, *Distributed Model Checking in SPIN*, Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, LNCS 1680, pp. 22-39, Springer, 1999.
31. A. Marowka, *Parallel Computing on Any Desktop*, Comm. of the ACM, 50 (9), pp. 75-78, September, 2007.
32. J.C. Philips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R.D. Skeel, L. Kale, K. Sculten, *Scalable Molecular Dynamics with NAMD*, Jornal of Computational Chemistry, 26:1781-1802, 2005.
33. U. Stern, D. Dill, *Parallelizing the Murϕ Verifier*, Proc. 9th Intl. Conf. Computer Aided Verification (CAV), LNCS 1254, pp. 256-278, Springer, 1997.
34. W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press, 1994.
35. A. Valmari, *The State Explosion Problem*, Lectures on Petri Nets I: Basic Models, LNCS Tutorials, LNCS 1491, pp. 429-528, Springer, 1998.