

Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking

Anton J. Wijs and Dragan Bošnački

Eindhoven University of Technology
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands

Abstract. We present several methods to improve the run times of probabilistic model checking on general-purpose graphics processing units (GPUs). The methods enhance sparse matrix-vector multiplications, which are in the core of the probabilistic model checking algorithms. The improvement is based on the analysis of the transition matrix structures corresponding to state spaces of a selection of examples from the literature.

Our first method defines an enumeration of the matrix elements (states of the Markov chains), based on breadth-first search which can lead to a more regular representation of the matrices. We introduce two additional methods that adjust the execution paths and memory access patterns of the individual processors of the GPU. They exploit the specific features of the transition matrices arising from probabilistic/stochastic models as well as the logical and physical architectures of the device.

We implement the matrix reindexing and the efficient memory access methods in GPU-PRISM, an extension of the probabilistic model checker PRISM. The experiments with the prototype implementation show that each of the methods can bring a significant run time improvement - more than four times compared to the previous version of GPU-PRISM. Moreover, in some cases, the methods are orthogonal and can be used in combination to achieve even greater speed ups.

1 Introduction

Probabilistic model checking (e.g. [18,2,3]) was introduced for the analysis of systems that contain inherently probabilistic components. It has been applied to a broad spectrum of systems, ranging from communication protocols, like FireWire and Bluetooth, to various biological networks.

Unlike in standard model checking, in probabilistic model checking the correctness of the verified properties is quantified with some probabilities. Such properties are expressed in special logics which are extensions of traditional temporal logics. As a result, probabilistic model checking algorithms overlap with conventional ones in the sense that they require computing reachability of the underlying transition systems. Still, there are also important differences because numerical methods are used to compute the probabilities.

Modern General Purpose Graphics Processing Units (GPUs) are no longer dedicated only to graphics applications. Instead a GPU can be seen as a general purpose manycore processor. The idea to use GPUs for model checking in general, and for probabilistic model checking in particular, was put forth in [8,9]. The main goal was to speed up the numerical components of the algorithms. More precisely, it turned out that one can harness the massively parallel processing power of GPUs to accelerate linear algebraic operations, like sparse matrix vector multiplication (SpMV) and its derivatives, which are at the core of the algorithms. Significant speed ups, often of more than ten times in comparison to the sequential analogues, can easily be achieved.

In this paper, we describe three novel methods to improve SpMV and related algorithms. The methods exploit the specific structures of the matrices that arise in probabilistic model checking. The matrices contain transition probabilities for the underlying Markov chains, which are actually the state spaces of the probabilistic models. Therefore we first present an overview of the transition matrices/state spaces based on the examples that occur in the standard distribution of the probabilistic model checker PRISM [17].

The efficiency of the GPU computations crucially depends on the usage of the various types of memory that are on the device. The difference in speed between various memories can be up to 100 times. Therefore, we strive to achieve so called *coalesced* memory access, i.e. the active processors of the GPUs fetch data from addresses which are physically close to one another. It turns out that to obtain such efficient memory access patterns it is advantageous to have elements of the matrix grouped as close as possible to the main diagonal. To achieve this we develop a heuristic that assigns indices to the states of the Markov Chains based on breadth-first search.

We also present two new SpMV methods, each consisting of a new matrix storage format and accompanying SpMV algorithm. These are geared towards maximizing coalesced memory access, in particular for matrices stemming from probabilistic model checking problems.

In the first method, each thread processes one row of the matrix. The algorithm groups the threads in segments of rows that conform nicely with the logical and physical architecture of the GPU. The specially tailored storage format ensures efficient access to contiguous memory locations. The second method also groups the rows in segments, with the difference that each row is processed by two threads working in parallel. Again, its specific storage format ensures coalesced memory access of all threads accessing a row in the same segment.

We implemented the new methods in GPU-PRISM [10], an extension of the probabilistic model checker PRISM. Each of the efficient memory access methods can achieve runtime improvements with regard to the previous version of GPU-PRISM of at least factor 2, and improvements of factor 4.5 have been recorded too.

2 GPU Preliminaries

Harnessing the power of GPUs is facilitated by specific Application Programming Interfaces. In this paper, we assume a concrete NVIDIA GPU architecture and the Compute Unified Device Architecture (CUDA) interface [13]. Nevertheless, the algorithms that we present here can be straightforwardly extended to a more general context, i.e., for an architecture which provides massive hardware multithreading, supports the single instruction multiple thread (SIMT) model, and relies on coalesced access to the memory.

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, run time API for memory management (allocation, deallocation), and statements to launch functions on GPU. In this section we give only a brief overview of CUDA. More details can be found in, for instance, [8].

CUDA Programming Model. A CUDA program consists of a *host* program which runs on the Central Processing Unit (CPU) and a CUDA *kernel*. The kernel, which describes the parallel part of the program, is executed many times in parallel by different threads on the GPU device, and is launched from the host program, which comprises the sequential part. Each GPU thread executes the same code. GPU threads are grouped in blocks. Each thread block 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 kinds of memory. Each thread has its own on-chip registers and off-chip local memory, which is quite slow. Threads within a block cooperate via shared memory which is on-chip and very fast. If multiple blocks 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 6GB), but slow since, like the local memory, it is not cached. The host (CPU program) has read and write access to the global memory (Video RAM, or VRAM), but cannot access the other memories (registers, local, shared). Thus, the global memory is used for communication between the host and the kernel.

CUDA Execution Model. A GPU 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). The threads of a block are executed in groups of 32 called *warps*. The threads in a warp execute instructions in a synchronous manner. These instructions can be different, but if they are all the same, the runtime is positively affected. Hence, our goal is to avoid execution divergence, i.e., to make the threads perform the same instructions as long as possible. Memory accesses of the threads in a single warp are also done simultaneously whenever possible,

and if these accesses can be grouped together physically, i.e. if the accesses are coalesced, this greatly improves the runtime. For this reason, achieving as much coalesced memory access as possible in SpMV is the main objective throughout this paper. We developed algorithms where the memory accesses of threads in a warp can be grouped together.

3 Structures of Transition Probability Matrices and BFS Reindexing

To exploit the specifics of the transition matrices that arise in probabilistic model checking, we analyze some case studies from the literature. In particular, we consider the examples of probabilistic and stochastic models that are part of the standard distribution of PRISM. 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- (DTMC) and continuous-time Markov chain (CTMC), respectively. For the sake of simplicity in this paper our focus is on discrete-time Markov chains (DTMC), so we opted for consistently using the qualification “probabilistic”. Nevertheless, the concepts and algorithms that we present here can be applied as well to continuous-time Markov chains. For the time being we do not consider models of the Markov decision processes type.

Since PRISM is probably the most widely applied probabilistic model checker, the examples that we consider give a fair representation of models that are used in applications. There are models from different areas, like probabilistic algorithms, queuing theory, chemistry, and biology.

Our first goal is to structurally examine the state spaces. Therefore, we make plots of the corresponding transition probability matrices. The existence of a probability greater than zero, i.e., a transition in the underlying Markov chain represented by the matrix element, is represented with a dot. The plots of the transition matrices are given on the left hand side of each pair of plots in Figures 2 and 3. Such plots can help identifying patterns in the elements which could be exploited in the algorithms.

In PRISM each state is given a number between 0 and $n - 1$, where n is the number of states in the underlying Markov chain. The plots on the left-hand side are based on the original indexing of the states as it is produced by (GPU-)PRISM. We explain below the plots on the right-hand side.

One can observe that there is often some regularity in the distribution of the non-zero elements. In most of the examples one can notice diagonal grouping of the elements. The diagonals are either parallel to the main matrix diagonal or they close some angle with it. The most notable in that regard are `cluster`, `tandem`, `cell`, and `molecules`, but also in the other examples (except `herman`) the diagonal structure is prevailing. The most remarkable of all is the matrix for

herman which has some sort of “fractal” structure, reminiscent of the Sierpinski carpet or similar examples.¹

3.1 Breadth-First Search Reindexing of the States

A diagonal grouping, similar to the one exhibited by the PRISM examples, has been exploited before in algorithms for SpMV to improve the runtimes [6,7,20]. This was based on the advantageous memory access pattern which arises from the distribution of the non-zero elements. Because of the diagonal structure, threads that belong to the same block access locations in the main memory which are close to each other. In the coalesced access the threads (preferably of the same block) access consecutive memory locations. This minimizes the number of accesses that are needed to provide data to all threads in the block. In the ideal case, all necessary data can be fetched simultaneously for all threads in the block.

For illustration, consider matrix M given in Fig. 1a in which the non-null and null elements are denoted with \bullet and \circ , respectively.

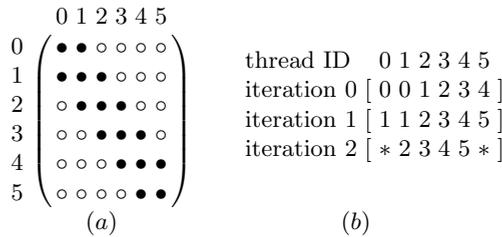


Fig. 1. (a) An example of a diagonally shaped matrix M . (b) A memory access pattern corresponding to the matrix.

We want to multiply M with a vector x . For simplicity, suppose that we use a kernel with a one dimensional grid. The grid consists of one block that contains six threads. Further, let each thread process one row in the multiplication algorithm by performing the inner product of the row with the vector. We assume that thread IDs range from 0 to 5 and that thread i processes row i , for $0 \leq i \leq 5$.

During the execution, we can observe the memory access pattern given in Fig. 1b. The top row of the pattern contains the thread IDs. The rest of the rows represent the access to the vector elements during the computation of the matrix vector product. Each of these rows corresponds to an iteration. In each row, the entry in a column corresponding to thread i contains the index of

¹ It would be worth investigating where this structure comes from and if there are also other examples of Markov chains, not necessarily in probabilistic model checking, that have this kind of a “fractal” structure. Considering that the fractals have been used for image compression, maybe one could develop an efficient compact representation of the transition matrices.

the element of vector x that is accessed in the iteration corresponding to the row. The special entry “*” denotes that the corresponding thread accesses no element during the iteration. In one iteration the thread k , corresponding to row k , computes the product of some non-zero element $M[k, l]$ of row k with element $x[l]$. For example, during iteration 0, both thread 0 and thread 1 access $x[0]$. For $2 \leq i \leq 5$, thread i uses $x[i - 1]$. Element $x[5]$ is not used during iteration 0. The other rows of the pattern are interpreted in an analogous way. One can see that in most of the cases threads with consecutive ID numbers access consecutive

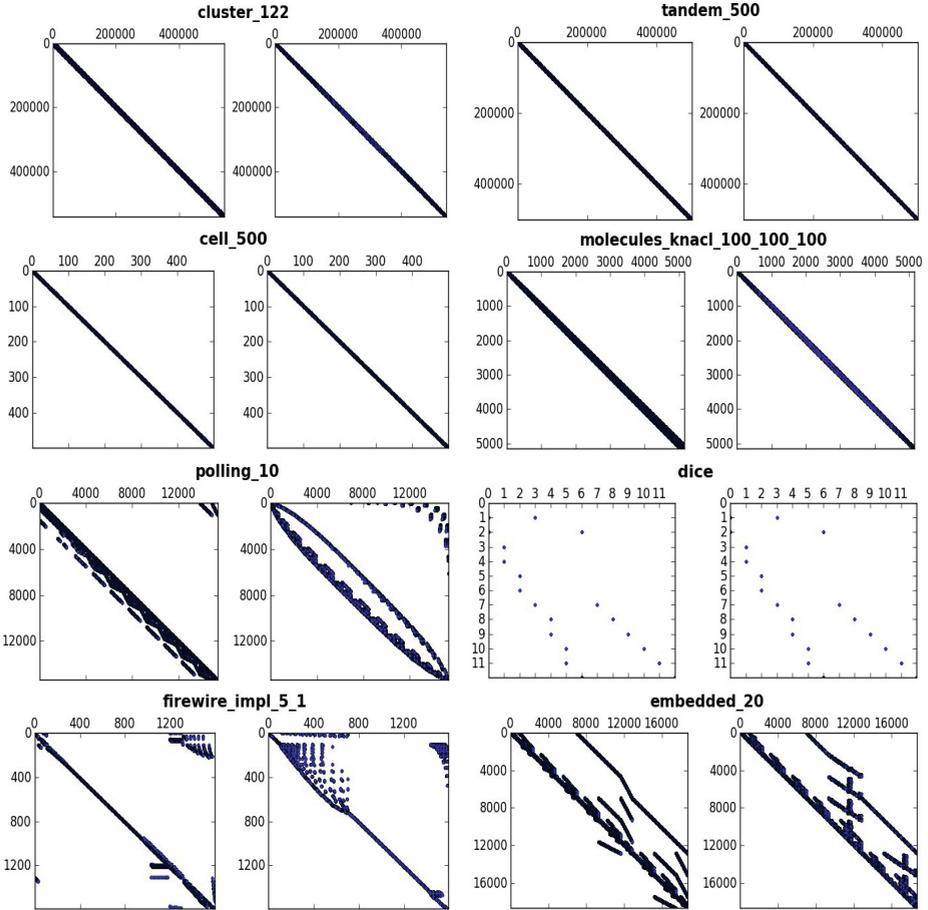


Fig. 2. Plots of transition matrices of models from the PRISM standard distribution. For each model two plots are given: the transition matrix produced by PRISM (left) and the transition matrix after the BFS reindexing (right). The numbers in the model names denote the values of the model parameters in the order they are asked by PRISM. The model names are (from left to right and top to bottom, respectively): `cluster`, `tandem`, `cell`, `molecules_knaci`, `polling`, `dice`, `firewire_impl`, and `embedded`.

indices – and therefore consecutive memory locations – that correspond to the elements of vector x .

However, the access to the memory locations corresponding to the matrix elements is not contiguous. As we show in Section 4, contiguous access can be achieved to a significant extent by using an appropriate memory storage format for the matrix.

Considering the potential benefits of the diagonal structure, a natural idea is to try to permute the indices of the matrix such that a diagonal structure is obtained. The approach that we use for that purpose is to re-enumerate the states of the underlying graph of the Markov chain in breadth-first search (BFS) order. The rationale behind this is to exploit the locality of the Markov chains, i.e., the fact that most of the states are connected to their immediate neighbors and that there are not too big transition “jumps” between states. This would ensure that the differences between the row and column indices of the non-zero elements of the matrix stay within a predefined interval, i.e., that they stay within some relatively narrow strip around the main diagonal.

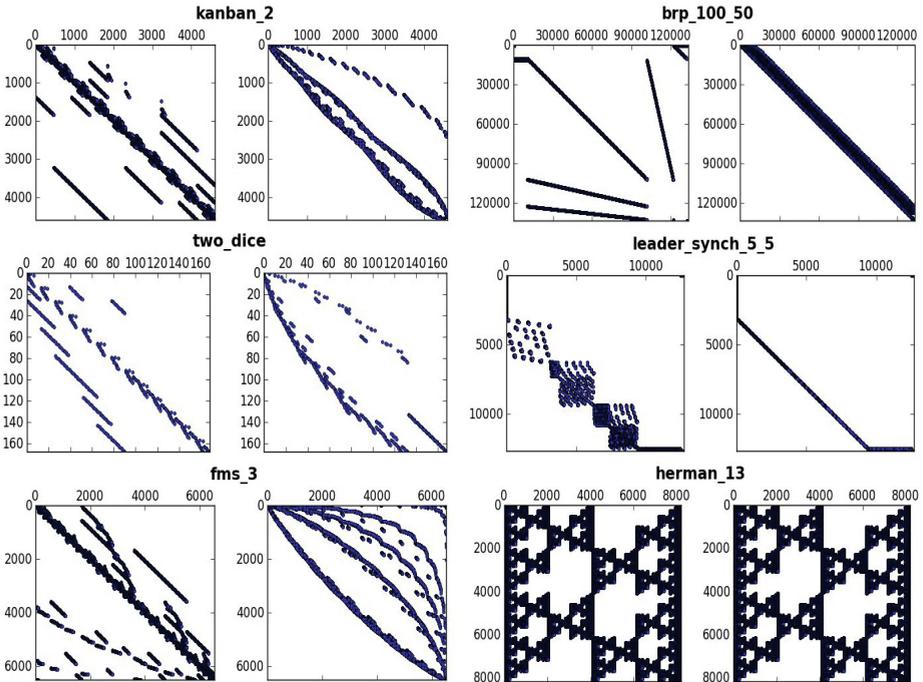


Fig. 3. (continued) Plots of transition matrices of models from the PRISM standard distribution. For each model two plots are given: the transition matrix produced by PRISM (left) and the transition matrix after the BFS reindexing (right). The numbers in the model names denote the values of the model parameters in the order they are asked by PRISM. The model names are (from left to right and top to bottom, respectively): `kanban`, `brp`, `two_dice`, `leader_synch`, `fms`, and `herman`.

The plots of the matrices after the BFS reindexing are given on the right-hand sides in Figs. 2 and 3. At least in two cases (**brp** and **leader**) the structure of the matrix has been “diagonalized”, in the sense that different lines/diagonals are brought closer to each other. In the case of **leader**, the original “staircase” structure is transformed into a line parallel to the main diagonal. The matrices of **kanban** and **two_dice** have become more compact, in the sense that there are fewer “islands” in the state space. Such a grouping of the indices positively affects memory access in SpMV, similar to clustering around the main diagonal. Matrices such as **cluster**, **tandem**, **cell**, and **polling**, which already had a diagonal structure, maintained it. Finally, the “fractal” example, **herman**, stays the same under reindexing, as well as the small example **dice**.

4 Coalescing Matrix Data Access

As we saw in the previous section, by grouping the non-zero elements of the matrix in diagonal shapes, a contiguous access to the elements of the vector is made possible. However, to actually achieve this in practice, the matrix should be stored in a suitable way in the memory of the GPU. Once we have convenient storage formats, corresponding algorithms must be developed that can efficiently exploit these new data structures. In this section, we present two new storage methods and their accompanying algorithms.

4.1 Sparse Matrix Representation

The storage size of a matrix is $O(n^2)$, where n is the number of rows. Sparse matrices, however, can be significantly compressed. Matrix compression is a standard technique used for probabilistic model checking. For this, special matrix storage formats are used. In this section, we build on the so-called *modified sparse row/column format* (MSR) [16,5]. We illustrate this format with the example in Fig. 4.

The non-zero elements of the matrix are linearly stored in the array *non-zeros*. Elements belonging to the same row are stored in consecutive cells. The

	0	1	2	3	4	5	6
0	0	a	b	0	0	0	0
1	0	0	c	d	0	0	0
2	0	0	0	0	e	0	0
3	f	0	0	g	0	0	0
4	0	0	0	0	h	0	0
5	i	j	0	0	0	0	0
6	0	0	k	0	0	0	0

<i>row-starts</i>	0	2	4	5	7	8	10	11			
<i>cols</i>	1	2	2	3	4	0	3	4	0	1	2
<i>non-zeros</i>	a	b	c	d	e	f	g	h	i	j	k

Fig. 4. An example of the MSR storage format. The letters denote the non-zero elements of the matrix. On the right-hand side is the MSR representation of the matrix.

beginning of each row is given by the array *row-starts*. Array *cols* contains the column indices of the corresponding elements in *non-zeros*.

Algorithm 1 is the basic kernel of an SpMV algorithm that is executed by each of the threads. This kernel was developed based on the sequential implementation of PRISM (cf. [8,9]).

In this paper, we only present the kernels, i.e., the parts of the algorithms that are executed on the GPUs, since the host program, the CPU parts, are fairly standard. A generic host program can be found in its integral form in our previous papers on GPU model checking [8]. Such a host program can be used as is for all algorithms presented in this section.

Algorithm 1. Standard SpMV Kernel for MSR Matrices

Require: *row-starts*, *cols*, *non-zeros*, *n*, *x*, *x'*, *BlockId*, *BlockSize*, *ThreadId*

```

1:  $i := \text{BlockId} \cdot \text{BlockSize} + \text{ThreadId}$ ;
2: if ( $i < n$ ) then
3:    $d := 0$ ;
4:    $l := \text{row-starts}_i$ ; // start of row
5:    $h := \text{row-starts}_{i+1}$ ; // end of row
6:   for ( $j = l; j < h; j++$ ) do
7:      $d := d + \text{non-zeros}_j \cdot x_{\text{cols}_j}$ ;
8:    $x'_i := d$ ;
```

Algorithm 1 assumes the MSR memory storage format. Therefore, the input is an MSR representation (in three separate arrays) of a matrix, followed by the matrix size *n*, the vector *x*, as well as the GPU bookkeeping IDs, i.e. the ID of the current thread and the block to which the thread belongs, and the size of each block. Vector *x'*, which is the output of the algorithm, is the result of the matrix-vector multiplication.

In line 1, the ‘absolute’ thread ID is computed since *ThreadId* is relative to the block. Variable *i* is used to determine the row that is processed by the thread. Line 2 is just a check whether this row number is within the matrix bounds. Variable *d* contains the temporary value of the inner product sum of the row *i* with vector *x*. In lines 4 and 5, we determine the start and the end, respectively, of the non-zero elements in *non-zeros* belonging to the designated row. The iteration in lines 6 and 7 computes the inner product which is stored in *d* and eventually assigned, in line 8, to the *i*-th element of the result *x'*.

A drawback of Algorithm 1 in combination with the MSR format is that, when the former is executed by the threads of a given block in parallel, the elements of array *non-zeros*, which are accessed by the threads, are not stored in consecutive memory locations. In the above example, assume a block size 4. Threads 0, 1, 2, and 3 of block 0, in their first iteration, need access to the elements of *non-zeros* which are the first elements of the corresponding rows. These are the elements *a*, *c*, *e*, and *f*, (at positions 0, 2, 4, and 5), respectively. As a result of such a non-contiguous access, several cycles might be needed to fetch all elements of

non-zeros. In contrast, if the elements were in consecutive positions, i.e., if they could have been accessed in a coalesced way, just a single access cycle would have been sufficient.

Note that this problem occurs also with the diagonally shaped matrices discussed in the previous section. Although the elements of x , which are processed in the same iteration by the threads of the same block, can be accessed in a coalesced way, this is still not the case with the *non-zeros* elements, i.e., with the standard MSR format.

4.2 A Full-Warp SpMV Algorithm

To coalesce the memory accesses in SpMV, we focus on the fact that the GPU groups the launched threads into warps. If threads in the same warp can access the memory in a coalesced way, data fetching will be done for all those threads in a single cycle.

To achieve coalesced access of the elements in a matrix within a warp of threads, we reorder the MSR representation of the matrix such that the elements accessed in a warp are next to each other. First of all, to explicitly group the threads in warps, we introduce a new array named *seg-starts*, which partitions the matrix into segments, each containing as many consecutive rows as the warp size (apart from the last segment, possibly). Say the warp size is 4, then the example given earlier will now be rewritten as given below. The double vertical lines indicate the warp boundaries. Note that some “dummy” elements need to be added to keep the elements of the same row on equidistant intervals. However, as we will see later in the experiments, this increase of memory is usually of no significance, and it is amply compensated by improved runtimes.

<i>seg-starts</i>	0	8	14											
<i>cols</i>	1	2	4	0	2	3	-	3	4	0	2	-	1	-
<i>non-zeros</i>	<i>a</i>	<i>c</i>	<i>e</i>	<i>f</i>	<i>b</i>	<i>d</i>	0.0	<i>g</i>	<i>h</i>	<i>i</i>	<i>k</i>	0.0	<i>j</i>	0.0

To exploit the modified matrix storage format, we introduce Algorithm 2. The new algorithm is a modification of Algorithm 1 and features the same input and output, except for the fact that the matrix dimension n is replaced by two numbers n_s and n_{rem} . The former is the predefined number of segments, i.e. n divided by the number of threads in a warp, whereas n_{rem} is the number of rows in the last segment. The last segment does not necessarily have the same size as the other segments, depending on whether or not n is divisible by the warp size. If we assume for our running example matrix that we have just one block and a warp size 4, then this will result in $n_s = 2$ and $n_{rem} = 3$.

Like in Algorithm 1, we begin by computing the ‘absolute’ thread ID, which also determines the index of the processed row. Besides that, in line 2 the segment ID *segid* is computed. As mentioned above, for our running example we will have two segments. In line 3, *lane* is computed which is an index of the thread within the warp, or in our case, since the warp and segment size are the same, it is also an index within the segment. In line 4 the matrix dimension n is recovered from

Algorithm 2. SpMV Kernel for MSR Matrices reordered into warp segments

Require: $seg\text{-}starts$, $non\text{-}zeros$, n_s , n_{rem} , x , x' , $BlockId$, $BlockSize$, $ThreadId$

```

1:  $i := BlockId \cdot BlockSize + ThreadId$ ;
2:  $segid := i / WarpSize$ ; // segment index
3:  $lane := ThreadId \ \& \ (WarpSize - 1)$ ; // thread index in warp
4:  $n = (n_s - 1) \cdot WarpSize + n_{rem}$ ;
5: if ( $i < n$ ) then
6:    $d := 0$ ;
7:   if  $segid < n_s - 1$  then // determine segment size
8:      $skip := WarpSize$ ;
9:   else
10:     $skip := n_{rem}$ ;
11:     $l := seg\text{-}starts_{segid}$ ; // start of segment
12:     $h := seg\text{-}starts_{segid+1}$ ; // end of segment
13:    for ( $j = l + lane; j < h; j = j + skip$ ) do
14:       $d := d + non\text{-}zeros_j \cdot x_{cols_j}$ ;
15:     $x'_i := d$ ;

```

the input values n_s and n_{rem} . The next difference compared to Algorithm 1 is in lines 7-10. This is because, unlike in the original MSR format, in the new format the *non-zeros* elements, belonging to the same row (and therefore are accessed by the same thread), are not stored contiguously. Instead they are dispersed regularly in the *non-zeros* array, i.e., separated by equal skip intervals. Note that in line 8 the skip for the last segment is set to n_{rem} , to take the possibly different size of the last segment into account. The start and end of the **for** loop are computed in lines 11 and 12, respectively, and they coincide with the start and end of the segment containing row i . The **for** loop in line 13 is started with offset $lane$ to take into account the relative position of the thread within the segment and the loop counter j is increased with step $skip$ to ensure that each thread i fetches the elements of row i .

One can see that for our running example with one block and two segments of size 4, threads 0, 1, 2, and 3 of the first segment will access in the first iteration through the **for** loop the first four elements of *non-zeros*, a , c , e , and f , respectively, i.e. the first elements of rows 0, 1, 2, and 3.

4.3 A Half-Warp SpMV Algorithm

The same coalescing approach can be used to obtain a matrix representation supporting multiple threads per row. If we set the segment size to half the warp size, assigning a warp of threads to each segment allows us to use two threads per row. This should be taken into account when rewriting the MSR representation of a matrix, in order to ensure that the elements of rows in a warp are grouped in pairs, as shown in the following example:

<i>seg-starts</i>	0	4	8	12	14									
<i>cols</i>	1	2	2	3	4	-	0	3	4	-	0	1	2	-
<i>non-zeros</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	0.0	<i>f</i>	<i>g</i>	<i>h</i>	0.0	<i>i</i>	<i>j</i>	<i>k</i>	0.0

Algorithm 3. SpMV Kernel for MSR Matrices reordered into half warp segments

Require: *seg-starts*, *non-zeros*, n_s , n_{rem} , x , x' , *BlockId*, *BlockSize*, *ThreadId*

```

1: __shared__ volatile double shared[ThreadsPerBlock/2]; // to store results
2:  $i := \text{BlockId} \cdot \text{BlockSize} + \text{ThreadId}$ ;
3:  $\text{segid} := i / \text{WarpSize}$ ; // segment index
4:  $\text{lane} := \text{ThreadId} \ \& \ (\text{WarpSize} - 1)$ ; // thread index in warp
5:  $\text{row} := i / 2$  // row id
6:  $n = (n_s - 1) \cdot (\text{WarpSize} / 2) + n_{rem}$ ;
7: if ( $\text{row} < n$ ) then
8:    $d := 0$ ;
9:   if  $\text{segid} < n_s - 1$  then // determine segment size
10:      $\text{skip} := \text{WarpSize}$ ;
11:   else
12:      $\text{skip} := n_{rem} \cdot 2$ ;
13:    $l := \text{seg-starts}_{\text{segid}}$ ; // start of segment
14:    $h := \text{seg-starts}_{\text{segid}+1}$ ; // end of segment
15:   for ( $j = l + \text{lane}; j < h; j = j + \text{skip}$ ) do
16:      $d := d + \text{non-zeros}_j \cdot x_{\text{cols}_j}$ ;
17:   if  $\text{lane} \% 2$  then // determine thread id in row
18:      $\text{shared}[\text{ThreadId}/2] := d$ ;
19:   if  $!(\text{lane} \% 2)$  then // accumulate results
20:      $x'_{\text{row}} := d + \text{shared}[\text{ThreadId}/2]$ ;

```

Corresponding to the new storage format is the half-warp based Algorithm 3. This algorithm requires the same data as its full-warp counterpart, except that it is assumed that the matrix is stored in the “half-warp” storage format Algorithm 2. In line 1, array *shared*, which resides in the shared memory, is defined. Recall that the shared memory is accessible by all threads that belong to the same block and it is around two orders of magnitude faster than the main GPU memory in which both the matrix and the vector are stored. In this algorithm, the inner product of one row with the vector is done by two threads, so the final result should be the sum of the two partial sums produced by each of the thread. This requires communication between the threads assigned to the same row, and the *shared* array is used for this purpose.

The assignments in lines 2-4 are the same as in Algorithm 2, only this time, since two threads are processing one row, *i* does not correspond to the row index. We compute the latter in line 5, such that two threads are assigned to each row. The lines 6-16 are as in Algorithm 2, the only subtlety being that the segment size is halved, and the skip for the last segment is set to $n_{rem} \cdot 2$. The

main difference with Algorithm 2 is in lines 17-20. This piece of code actually checks if *lane*, the index of the thread within the segment, is even or odd. If the index is odd, then the end result (partial sum of the inner product) is saved in *shared* at the position corresponding to the row. Otherwise, the end result for the row is produced by adding the partial result from *shared* (of the other thread processing the same row) to the partial result obtained by this thread. There are no race conditions when accessing *shared*, since the threads in a warp execute their instructions in a synchronous manner, so the writing to *shared* is strictly done before the reading starts.

Again, one can see that the algorithm in combination with the matrix storage ensures coalesced accesses to the matrix elements of the threads within a segment.

5 Experimental Results

The BFS reindexing as well as the half and full-warp methods were implemented in GPU-PRISM 4.0 [10],² an extension of the model checker PRISM version 4.0. We conducted a number of experiments with our implementations on a 64-bit computer running Ubuntu 10.10 with CUDA version 4.1, both the Software Development Kit and the driver. It runs on an AMD Athlon(tm) 64 X2 Dual-Core Processor 3800+ running at 2 GHz with 4 GB RAM, and has an NVIDIA GPU GeForce GTX 480 with 1.5 GB global memory and 480 cores running at 1.4 GHz. As block size, we used 512 threads.

The data of the experiments were both represented in MSR format, and in the special *compact* MSR (CMSR) format [16], which was specifically designed to efficiently store matrices representing probabilistic models. These matrices tend to be not only sparse, but also contain a relatively small number of distinct values. This is exploited in CMSR by keeping these values in a separate array, and storing pointers to these values, instead of the values themselves, in the *non-zeros* array. In [16], it is remarked that besides memory benefits, CMSR also tends to speed up the computations, due to caching effects. Intuitively, in the GPU setting, the use of the CMSR format instead of the MSR format reduces the potential for coalesced memory access; the best one can do is reorder the pointers to the values, not the values themselves. Since CMSR is used by default in PRISM, and SpMV on a CPU with the CMSR format usually outperforms SpMV with MSR, it is crucial that we test the efficiency of the half- and full-warp methods with CMSR, as well.

All models that we used in our experiments were taken from the standard distribution of PRISM. Table 1 shows the common characteristics of the experiments. The first and the second column, respectively, contain the name and the instance (depending on the parameter values) of the model. The third column denotes the number of the property in the property file that comes with each model. The last two columns give the number of reachable states and the number of iterations required to solve the system of linear equations represented by the combination of the model and the property to check, using the Jacobi method.

² <http://www.win.tue.nl/~awijs/software.html>

Table 1. Information on the protocol properties

Model	Inst.	Prop.	n	Iterations
herman	15/5	3	32,768	245
cluster	320	1	3,704,340	5,107
cluster	464	1	7,776,660	23,932
tandem	1,023	1	2,096,128	16,326
tandem	2,047	1	8,386,560	24,141
kanban	5	1	2,546,432	663
fms	7	1	1,639,440	1,258
fms	8	1	4,459,455	1,438
polling	17	4	3,342,336	4,732
polling	18	4	7,077,888	4,880

Table 2 presents the results obtained when using standard GPU SpMV on both the original MSR matrices, as produced by PRISM, and the BFS-reindexed ones. As in the previous table, the first two columns give the name and instance of the model. The next column gives the consumed memory which is the same in both cases, since the data storage format is unchanged. Columns 4 and 5 contain the times with the PRISM and reindexed matrix, respectively. The last column gives the speed up factor which is obtained by dividing the original time with the time obtained with the reindexed matrix.

Table 2. Performance of standard SpMV on MSR and BFS-reindexed MSR data

Model	Inst.	mem.	orig. time	+BFS time	Factor
herman	15	165	15.50	12.46	1.24
cluster	320	305	45.45	44.79	1.01
cluster	464	642	440.16	443.06	0.99
tandem	1,023	139	39.56	43.91	0.90
tandem	2,047	559	228.18	255.57	0.89
kanban	5	347	14.78	15.34	0.96
fms	7	198	15.18	15.08	1.01
fms	8	560	52.14	50.28	1.04
polling	17	295	77.25	66.21	1.17
polling	18	646	184.12	160.77	1.15

In most of the cases there is some speed up which is probably due to the coalesced access to the vector elements. On the other hand, the best result is achieved for an instance of **herman**, which has the ‘fractal’ structure and it is invariant under the reindexing. This could be due to the fact that during the reindexing, the matrix and the correspondingly permuted vector are copied to a new data structure. Although the new structures are conceptually identical to the original MSR-based structures of PRISM, they might provide a faster memory access. Obviously, a more thorough analysis is needed to explain this

Table 3. Performance of SpMV(WL) and SpMV(HWL) on MSR data

Model	Inst.	Original matrix					BFS reindexed matrix				
		SpMV(WL)		SpMV(HWL)		Factor (max.)	SpMV(WL)		SpMV(HWL)		Factor (max.)
		mem.	time	mem.	time		mem.	time	mem.	time	
herman	15	692	9.90	520	3.43	4.51	692	5.60	520	3.43	4.52
cluster	320	372	21.47	386	26.52	2.12	320	18.35	434	25.50	2.48
cluster	464	781	211.65	811	259.69	2.08	669	178.94	909	247.81	2.46
tandem	1,023	132	27.18	144	24.73	1.60	144	29.90	192	41.00	1.32
tandem	2,047	528	159.38	577	96.15	2.37	576	172.07	769	234.40	1.32
kanban	5	384	2.98	390	3.29	4.99	406	3.25	467	3.52	4.55
fms	7	248	3.88	242	4.30	3.91	261	3.93	261	4.30	3.86
fms	8	700	12.87	684	13.82	4.05	746	12.73	745	13.43	4.10
polling	17	329	20.43	329	23.31	3.78	496	25.86	505	30.42	2.99
polling	18	717	46.62	718	51.77	3.95	1,090	58.63	1,110	67.79	3.14

phenomenon. In general, although the results are not conclusive, it seems that the reindexing itself is able to produce some modest speed up.

Table 3 shows the results when applying the new algorithms using row segments to coalesce memory access. SpMV(WL) and SpMV(HWL) denote the algorithms with full- and half-warp segment size, respectively. For both algorithms, the memory use in megabytes, and run time in seconds are shown. The last column contains the maximal speed up factor with respect to the standard GPU-PRISM (without BFS reindexing), which can be found in Table 2.

For the original matrices, again the best speed up of 4.51 is obtained with **herman**, but this time this is paired with around the same factor of memory increase. The speed ups with the other models though are achieved with quite acceptable price in memory. It is important to note that the half-warp algorithm produces the best results only for the **herman** case; in all other cases the full-warp algorithm is the fastest. The **herman** matrices are relatively dense compared to the others, which supports the observation in related work, e.g. [6], that further parallelisation of individual row-vector multiplications, i.e. using multiple threads per row, often does not pay off for sparse matrices. In [6], this is related to assigning warps to rows, but here, even two threads per row does not outperform one thread per row, when typical sparse matrices of probabilistic models are used.

Table 3 also contains the results when using combinations of the new algorithms, i.e. first reindexing the matrix using BFS, and then partitioning the rows into segments. One can see that the results with **herman** are unaffected by the reindexing. This is in accord with our intuition since the transition matrix of this model is virtually invariant under the BFS reindexing. The results for **cluster** show that with the full-warp version of the algorithm, the BFS reindexing results in some gain in memory. Also the reindexing results in some additional speedup. For the other examples though, the reindexing causes deterioration of both the speed ups and the memory usage, suggesting that BFS reindexing is a technique which does not combine well with the improved SpMV methods.

Table 4. Performance of standard SpMV on CMSR and BFS-reindexed CMSR data

Model	Inst.	mem.	orig. time	+BFS time	Factor
herman	15	55	8.70	8.62	1.01
cluster	320	146	20.81	19.66	1.06
cluster	464	308	203.05	197.19	1.03
tandem	1,023	71	22.77	23.77	0.98
tandem	2,047	287	124.17	135.17	0.92
kanban	5	146	4.81	5.20	0.93
fms	7	86	5.75	5.76	1.00
fms	8	240	19.70	19.13	1.03
polling	17	189	35.65	41.44	0.86
polling	18	414	80.43	96.29	0.84

Table 5. Performance of SpMV(WL) and SpMV(HWL) on CMSR data

Model	Inst.	Original matrix					BFS reindexed matrix				
		SpMV(WL)		SpMV(HWL)		Factor	SpMV(WL)		SpMV(HWL)		Factor
		mem.	time	mem.	time	(max.)	mem.	time	mem.	time	(max.)
herman	15	231	5.48	173	3.56	2.44	692	5.60	520	3.43	2.54
cluster	320	159	17.05	164	24.57	1.22	142	15.09	152	23.93	1.38
cluster	464	335	175.40	346	255.72	1.16	298	162.17	319	253.89	1.25
tandem	1,023	64	23.77	68	39.34	0.96	68	25.22	68	39.85	0.90
tandem	2,047	256	139.42	273	230.46	0.89	272	146.78	273	231.25	0.85
kanban	5	152	2.18	154	2.84	2.21	159	2.37	161	3.05	2.03
fms	7	98	3.03	96	3.89	1.90	102	3.08	102	3.87	1.87
fms	8	276	9.92	271	12.64	1.99	291	9.69	291	12.06	2.03
polling	17	208	18.32	209	26.63	1.95	204	19.37	207	27.50	1.84
polling	18	455	37.89	456	58.48	2.12	447	42.89	453	58.16	1.88

Tables 4 and 5 show the results for the same model instances as Tables 2 and 3, but now using the CMSR data storage format. As expected, overall, the achieved speedups are not as high as when using MSR. BFS reindexing even shows a negative effect in combination with standard SpMV. It seems that the reindexing disturbs the optimization introduced by the CMSR storage format. Further research is required to determine the exact cause. The full-warp algorithm, however, still produces in most cases a speedup of two times. For the models `cluster` and `tandem`, it does not result in a speedup, which seems to be related to the fact that their matrices are perfect diagonals, and therefore probably already lead to relatively coalesced data access in SpMV. Finally, as when using MSR, the half-warp algorithm only outperforms the full-warp algorithm for the `herman` case.

6 Conclusions, Prospects and Related Work

We gave an overview and analysis of the state spaces that arise in probabilistic model checking, as represented by their corresponding transition probability matrices. Most of them show regular patterns and diagonally shaped matrices are prevailing. Based on this analysis, we suggested three methods for improving the run times of the model checking algorithms. All methods were implemented in GPU-PRISM, an extension of the probabilistic model checker PRISM.

Our first method performs a BFS-based reindexing of the states, which potentially leads to more compact representations of the matrices. The experiments with our implementation show that for some models the BFS reindexing can accelerate the model checking algorithms on GPUs.

Additionally, we proposed two methods that group the threads in segments. By choosing the segment size to coincide with a full or half-warp size, together with appropriate modifications of the data representation, one can achieve a coalesced access to the main GPU memory. The experiments showed that in some cases the model checking algorithms can be accelerated more than four times. Also combinations of the two coalescing methods with the BFS reindexing can produce some additional speed ups, but in most cases, the two techniques do not agree.

We intend to perform more experiments with different models from the PRISM set of examples as well as from other sources. It would also be worthwhile to further investigate the state spaces structures. A special challenge in that direction could be the fractal-like structures which were observed in one of the examples. These can potentially be used to optimize the storage of the state spaces as well as the run times of the algorithms.

Related work. GPU model checking was a logical continuation of the concept of multi-core model checking [15]. Besides the above mentioned introductory papers on GPU (probabilistic) model checking [8,9,10], several algorithms for SpMV, which exist in the literature, were recently tested in the context of probabilistic model checking [12]. This work complements our previous work in [8,9] to a significant extent. The paper seems to confirm our hypothesis presented there that our algorithms for GPU probabilistic model checking from [8] are superior to the class of Krylov methods, representatives of which were tested in [12].

An overview of algorithms for SpMV can be found in [20]. Several methods for SpMV were discussed in [6,7]. Among them are methods for diagonally shaped sparse matrices, which could play an important rôle in probabilistic model checking. They consider algorithms which are analogous with our half-warp algorithm, in which several threads process one row. They conclude that this really gives results only if the matrices are dense. This is confirmed by our results with the half-warp algorithm. Often we do not get any improvement, even though a row is processed by only two threads. Compared to our work, they do not consider BFS reindexing, but the most important difference is that we group the rows in segments of one- and half-warp sizes, which is not the case in their work. Also our matrix and vector memory storage differs from the ones used by them.

In [19] the authors propose to speed up probabilistic model checking, by exploiting the structure of the underlying Markov chains, for sequential algorithms. It might be interesting to investigate a combination of the findings about the structure of the state spaces presented in this paper and theirs in the GPU context.

Previous algorithms for parallel probabilistic model checking were almost exclusively designed for distributed architectures, i.e., clusters [5,11]. They were focused on increasing the state spaces of the models instead of the run times and minimizing the communication overhead between the threads instead of the memory latency. In [1], a shared memory algorithm is introduced for CTMC construction, but the algorithms employed there are quite different from our approach.

There are other publications that deal with other kinds of model checking on GPUs that do not involve probabilities (e.g.,[4,14]). They use algorithms which are quite different from the ones presented in this paper, since they do not focus on parallelizing a numerical computation, but on parallelizing state space exploration.

References

1. Allmaier, S.C., Kowarschik, M., Horton, G.: State Space Construction and Steady-state Solution of GSPNs on a Shared-Memory Multiprocessor. In: Proc. 7th Intt. Workshop on Petri Nets and Performance Models, PNPM 1997, pp. 112–121. IEEE Comp. Soc. Press (1997)
2. Baier, C., Katoen, J.-P.: Principles of Model Checking, 950 p. MIT Press (2008)
3. Baier, C., Katoen, J.-P., Hermanns, H., Haverkort, B.: Model-Checking Algorithms for Continuous-Time Markov Chains. *IEEE Transactions on Software Engineering* 29(6), 524–541 (2003)
4. Barnat, J., Brim, L., Ceska, M., Lamr, T.: CUDA Accelerated LTL Model Checking. In: IEEE 15th International Conference on Parallel and Distributed Systems, ICPADS 2009, pp. 34–41. IEEE (2009)
5. Bell, A., Haverkort, B.R.: Distribute Disk-based Algorithms for Model Checking Very Large Markov Chains. In: Formal Methods in System Design, vol. 29, pp. 177–196. Springer (2006)
6. Bell, N., Garland, M.: Efficient Sparse Matrix-Vector Multiplication on CUDA NVIDIA. Technical Report NVR-2008-004 (December 2008)
7. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009. ACM (2009)
8. Bošnački, D., Edelkamp, S., Sulewski, D.: Efficient Probabilistic Model Checking on General Purpose Graphics Processors. In: Pășăreanu, C.S. (ed.) SPIN 2009. LNCS, vol. 5578, pp. 32–49. Springer, Heidelberg (2009)
9. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.J.: Parallel probabilistic model checking on general purpose graphics processors. *STTT* 13(1), 21–35 (2011)
10. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.J.: GPU-PRISM: An Extension of PRISM for General Purpose Graphics Processing Units (tool paper). In: Proc. Joint HiBi/PDMC Workshop, HiBi/PDMC 2010, pp. 17–19. IEEE Computer Society Press (2010)

11. Ciardo, G.: Distributed and Structured Analysis Approaches to Study Large and Complex Systems. European Educational Forum: School on Formal Methods and Performance Analysis, 344–374 (2000)
12. Cormie-Bowins, E.: A Comparison of Sequential and GPU Implementations of Iterative Methods to Compute Reachability Probabilities. In: Proc. of Workshop on GRAPH Inspection and Traversal Engineering, GRAPHITE 2012, Tallinn, Estonia, April 1 (to appear, 2012)
13. http://www.nvidia.com/object/cuda_home.html#
14. Edelkamp, S., Sulewski, D.: Model Checking via Delayed Duplicate Detection on the GPU, Technical Report 821, Universität Dortmund, Fachberich Informatik (2008) ISSN 0933-6192
15. Holzmann, G.J., Bošnački, D.: The Design of a multi-core extension of the Spin Model Checker. IEEE Trans. on Software Engineering 33(10), 659–674 (2007) (first presented at: Formal Methods in Computer Aided Design (FMCAD), San Jose, November 2006)
16. Kwiatkowska, M., Mehmood, R.: Out-of-Core Solution of Large Linear Systems of Equations Arising from Stochastic Modelling. In: Hermanns, H., Segala, R. (eds.) PROBMIV 2002, PAPM-PROBMIV 2002, and PAPM 2002. LNCS, vol. 2399, pp. 135–151. Springer, Heidelberg (2002)
17. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic Symbolic Model Checker. In: Field, T., Harrison, P.G., Bradley, J., Harder, U. (eds.) TOOLS 2002. LNCS, vol. 2324, pp. 200–204. Springer, Heidelberg (2002)
18. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic Model Checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007)
19. Kwiatkowska, M.Z., Parker, D., Qu, H.: Incremental Quantitative Verification for Markov Decision Processes. In: Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 359–370. IEEE (2011)
20. Vuduc, R.W.: Automatic Performance Tuning of Sparse Matrix Kernels. Ph.D. thesis, University of California, Berkeley (2003)