Improving GPU Sparse Matrix-Vector Multiplication for Probabilistic Model Checking

Anton Wijs and Dragan Bošnački

Eindhoven University of Technology, 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 different 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 implemented 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 the traditional temporal logics. As a result, probabilistic model checking algorithms overlap with the 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 for probabilistic model checking, in particular, was developed in [8,

9]. The main goal was to speed up the numerical components of the algorithms. More precisely, it turns out that one can harness the massively parallel processing power of the GPUs to accelerate linear algebraic operations, like sparse matrix vector multiplication (SpMV) and its derivatives, which are in 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 the sparse vector multiplication and the 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 model checker PRISM [17].

The efficiency of the GPU computations crucially depends on the usage of the various types of memories 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 to the memory, when 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 heuristics that assigns indices to the states of the Markov Chains based on breadth-first search. This preprocessing of the matrix is already sufficient to produce significant speed ups (up to five times) with the standard versions of the sparse matrix-vector multiplication algorithms.

We also present two additional algorithms that are based on the so-called coalesced access to the memory, when threads which are performed simultaneously request data from memory locations which are sufficiently close to each other.

In the first of these algorithms 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. This 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.

We implemented our methods in GPU-PRISM [10], an extension of the probabilistic model checker PRISM. Each of the efficient memory access methods can achieve runtime improvements up to factor 4.5.

2 GPU Preliminaries

Harnessing the power of GPUs is facilitated by the new APIs. In this paper we assume a concrete NVIDA GPU architecture and its 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 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 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-chip local memory, which is quite slow. Threads within a block cooperate via shared memory which is on-chip 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 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 Excution Model. 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*. All threads of the warp execute a single (not necessarily the same) instruction. Thus, each thread of the warp can basically execute its own program. However, our goal is to avoid such an execution divergence, i.e., to make the threads perform the same execution as long as possible. Also with regard to the memory access threads can also use different addresses which leads to a memory divergence. This should also be avoided and this is actually our main objective throughout this paper. We develop algorithms that request sufficiently close memory locations, such that requests of the same warp can be grouped together (coalesced) for a more efficient memory access.

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. Since PRISM is probably the most widely applied probabilistic model checker, these examples give a good overview about the kinds of models that are used in applications. There are models from different areas, like probabilistic algorithms, queuing theory, chemistry, and biology. (We do not consider models of the Markov decision processes type.)

Our first goal is to qualitatively examine the state spaces. Therefore, we make plots of the corresponding transition probability matrices. The existence of the 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 possibly 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]

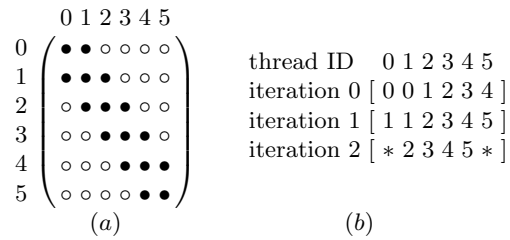
3.1 Breadth-first search reindexing of the states

A diagonal grouping, similar to the one exhibited by the PRISM examples, has been exploited before in algorithms for sparse matrix-vector multiplication to improve the run-times [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

[1] It would be worth exploiting 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.

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.



(a) (b)

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

We want to multiply M with a vector x. For simplicity, suppose that we use to this end a kernel with one dimensional grid. The grid consists of one block that contains six threads. Further, let one thread processes 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 the vector element that is accessed in the iteration corresponding to the row. The special entry "*" denotes that the corresponding thread accesses no element during the iteration. Which element of vector x is accessed by the thread is determined by the column index of the non-null element of the matrix which is processed by the thread during the corresponding iteration. Therefore, during iteration 0, both thread 0 and thread 1 access $x[1]$, 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 indices and therefore to consecutive memory locations that correspond to the elements of vector x.

The access to the memory locations corresponding to the matrix elements is not contiguous though. However, as we show in Section 4, this can also 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

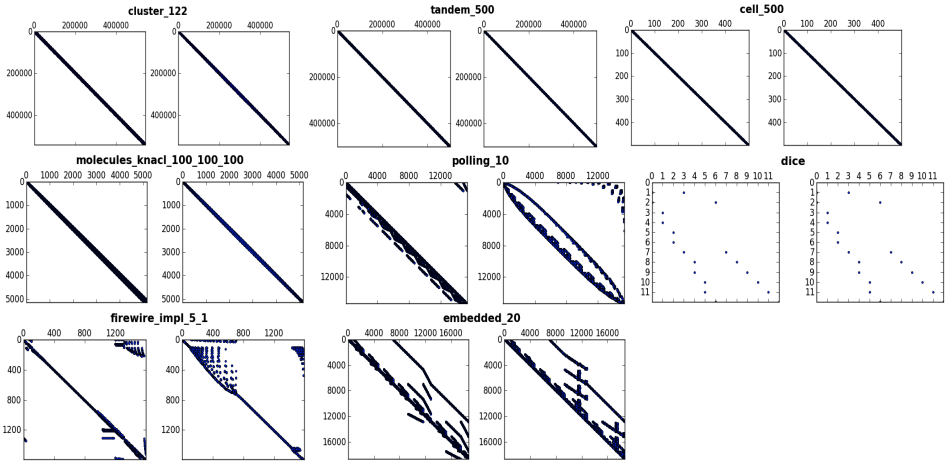


Fig. 2. Plots of transition matrices of models from the PRISM standard distribution. For each model two plots are given: the transition matrix produced by PRISM (left) and the transition matrix after the BFS reindexing (right). The numbers in the model names denote the values of the model parameters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

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 rational 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.

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 less "islands" in the state space. One can benefit also from such a grouping of the indices like for the clustering around the main diagonal, for analogous reasons. Moreover, in the matrices that had already a "nice" diagonal structure, like `cluster`, `tandem`, `cell`, and `polling`, the structure is preserved. The "fractal" example, `herman`, stays the same under reindexing as well as the small example `dice`.

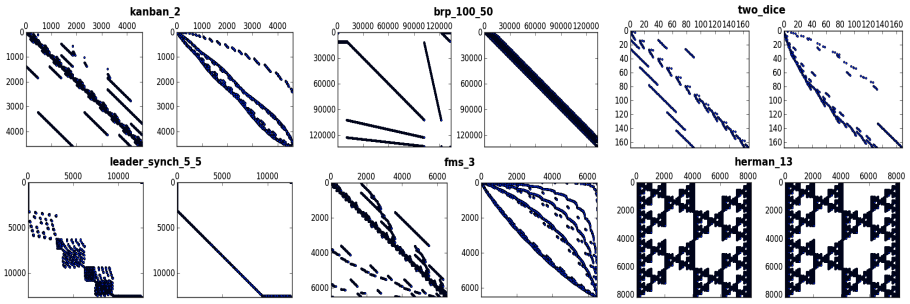


Fig. 3. (continued) Plots of transition matrices of models from the PRISM standard distribution. For each model two plots are given: the transition matrix produced by PRISM (left) and the transition matrix after the BFS reindexing (right). The numbers in the model names denote the values of the model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

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. To also achieve the same for the non-zero elements of the matrix, special care should be taken of the format in which the matrix is stored in the memory of the GPU. Also, once we have the convenient storage formats corresponding algorithms should be developed that can efficiently exploit the new data structure. In the sequel we present two new storage methods and their corresponding algorithms.

4.1 Sparse Matrix Representation

Although in theory the size of the matrix is in general $\Theta(n^2)$, where n is the number of rows, for sparse models that usually appear in practice the matrices can be significantly compressed. Such matrix compression 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 use the so called *modified sparse row/column format* (MSR) [16, 5] or its modifications. We illustrate this format on 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 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]).

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} 0 & a & b & 0 & 0 & 0 & 0 \\ 0 & 0 & c & d & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & e & 0 & 0 \\ f & 0 & 0 & g & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & h & 0 & 0 \\ i & j & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & k & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

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

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

In the sequel we give only 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]. A host program, which is essentially the same, can be used 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 := BlockId · BlockSize + ThreadId;
2: if (i < n) then
3:    d := 0;
4:    l := row-starts_i;                          // start of row
5:    h := row-starts_{i+1};                      // end of row
6:    for (j = l; j < h; j++) do
7:       d := d + non-zeros_j · x_{cols_j};
8:    x'_i := d;
```

Algorithm 1 assumes an MSR memory storage format. Therefore, input of the algorithm is an MSR representation (as a three separate arrays), followed by the matrix size n, vector x, as well as the GPU bookkeeping IDs. 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 also gives the number of the row that is processed by the thread. Line 2 is just a check if the 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, in the MSR representation of the segment which contains the elements of row i. The iteration in lines 6 and 7 computes the inner product which is stored in d and eventually assigned, in line 8, to i-th element of the result x'.

A drawback of Alg. 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 required by the threads, are not stored on 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 on a consecutive positions, i.e., if they could have been accessed in a coalesced way, just one 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.

4.2 A Full-Warp SpMV Algorithm

The parallel execution of the different row-vector multiplications can be further improved. For this, we exploit 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 its MSR representation 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 also later in the experiments, this increase of memory is usually of no significance and it is amply compensated by the improved run times.

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

To exploit the modified matrix storage format, we introduce Algorithm 2. The new algorithm is a modification of Alg. 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, whereas n_{rem} is the number of rows in a possibly "incomplete" block, i.e., if the number of rows n is not divisible by the warp size. The latter is assumed to be given. 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 Alg. 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

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

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

```
 1: i := BlockId · BlockSize + ThreadId;
 2: segid := i / WarpSize;                                  // segment index
 3: lane := ThreadId & (WarpSize − 1);                      // thread index in warp
 4: n = (n_s − 1) · 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-starts_segid;                               // start of segment
12:    h := seg-starts_segid+1;                             // end of segment
13:    for (j = l + lane; j < h; j = j + skip) do
14:       d := d + non-zeros_j · x_cols_j;
15:    x'_i := d;
```

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 the input values n_s and n_{rem}. The next difference compared to Alg. 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 therefor em, 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. Not that in line 8 the skip for the last block is set to n_{rem}. The start and end of the for iteration are computed in lines 11 and 12, respectively, and they coincide with the start and end of the segment which contains thread (row) i. The for iteration in line 13 is started with an 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 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 the first iteration the first four elements of *non-zeros*, a, b, c, and d, respectively.

4.3 A Half-Warp SpMV Algorithm

The same coalescing approach can be used to obtain a matrix representation supporting the use of segments whose number of rows is at most half the warp size. In that setting, assigning a warp of threads to each segment allows to use two threads per row. When rewriting the MSR representation of a matrix, we ensure that the elements of rows in a warp are grouped in pairs, as shown in the following example:

seg-starts	0	4	8	12	14									
cols	1	2	2	3	4	-	0	3	4	-	0	1	2	-
non-zeros	a	b	c	d	e	0.0	f	g	h	0.0	i	j	k	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 := BlockId · BlockSize + ThreadId;
 3: segid := i/WarpSize;                                           // segment index
 4: lane := ThreadId & (WarpSize − 1);                             // thread index in warp
 5: row := i/2                                                     // row id
 6: n = (n_s − 1) · (WarpSize/2) + n_rem;
 7: if (row < n) then
 8:    d := 0;
 9:    if segid < n_s − 1 then                                     // determine segment size
10:       skip := WarpSize;
11:    else
12:       skip := n_rem · 2;
13:    l := seg-starts_segid;                                      // start of segment
14:    h := seg-starts_segid+1;                                    // end of segment
15:    for (j = l + lane; j < h; j = j + skip) do
16:       d := d + non-zeros_j · x_cols_j;
17:    if lane % 2 then                                            // determine thread id in row
18:       shared[ThreadId/2] := d;
19:    if !(lane % 2) then                                         // accumulate results
20:       x'_row := d + shared[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 Alg. 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 a sum of the two partial sums produced by each of the thread. Thus each thread sets its end sum in a corresponding element of *shared*. The assignments in lines 2-4 are the same as in Alg. 2. Only this time, since two threads are processing one row, i does not correspond to the row index. So, we compute the latter in line 5. The lines 6-16 are like in Alg. 2. The only subtlety is that the segment size is halved, the skip for the last block is set to $n_{rem} \cdot 2$. The main difference with Alg. 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. In the ID is odd, then the end result (partial sum of the inner product) is saved in the corresponding element of

shared. Otherwise, the end result for the row is produced by adding the partial result from *shared* by the other thread which processes the same row.

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

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 of 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.

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

[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

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

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

For the original matrices, again the best speed up of 4.51 is obtained with `herman`, but this time this is paid 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

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.

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

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.

We intend to perform more experiments with different models from the PRISM set of examples as well as from other sources. Also it would be worth to explore further the structure of the state spaces. A special challenge in that direction could be the fractal-like structures which were observed in one of the examples. This can be potentially 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 [8–10], several algorithms for sparse matrix vector multiplication, which exist in the literature, were recently tested in the context of probabilistic model checking [12]. This work to a significant extent complements our previous work in [8, 9]. 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 sparse matrix-vector multiplication can be found in [20]. Several methods for sparse vector matrix multiplication were discussed in [6, 7]. Among them are also methods for diagonally shaped sparse matrices, which could play an important role 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. In a sense this is confirmed with our results with the half-warp algorithm. Often we do not get any improvement although a row is processed with 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 their in the GPU context.

Previous algorithms for 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 runtimes and minimizing the communication overhead between instead of the 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 different kinds of model checking on GPUs that do not involve probabilities (e.g.,[14, 4]). They use algorithms which are quite different from the ones presented in this paper.

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 Performance 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. J. Barnat, L. Brim, M. Ceska, T. Lamr, *CUDA Accelerated LTL Model Checking*, IEEE 15th International Conference on Parallel and Distributed Systems, ICPADS 2009, pp. 34-41, IEEE, 2009.
5. 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.
6. N. Bell, M. Garland, *Efficient Sparse Matrix-Vector Multiplication on CUDA* NVIDIA Technical Report NVR-2008-004, December 2008.
7. N. Bell, M. Garland, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, ACM, 2009.
8. D. Bošnački, S. Edelkamp, D. Sulewski, *Efficient Probabilistic Model Checking on General Purpose Graphics Processors* Proc. 16th International SPIN Workshop, LNCS 5578, pp. 32–49, Springer, 2009.
9. D. Bošnački, S. Edelkamp, D. Sulewski, A.J. Wijs, *Parallel probabilistic model checking on general purpose graphics processors*, STTT, 13:1,21-35, 2011.

10. D. Bošnački, S. Edelkamp, D. Sulewski, A.J. Wijs, *GPU-PRISM: An Extension of PRISM for General Purpose Graphics Processing Units (tool paper)*, Proc. Joint HiBi/PDMC Workshop (HiBi/PDMC'10), Enschede, pp. 17-19, IEEE Computer Society Press, 2010.
11. 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.
12. E. Cormie-Bowins, *A Comparison of Sequential and GPU Implementations of Iterative Methods to Compute Reachability Probabilities*, Proc. of Workshop on GRAPH Inspection and Traversal Engineering (GRAPHITE 2012), Tallinn, Estonia, April 1, 2012. (to appear)
13. `http://www.nvidia.com/object/cuda_home.html#`
14. S. Edelkamp, D. Sulewski, *Model Checking via Delayed Duplicate Detection on the GPU*, Technical Report 821, Universität Dortmund, Fachberich Informatik, ISSN 0933-6192, 2008.
15. 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.)
16. M.Z. Kwiatkowska, R. Mehmood, *Out-of-Core Solution of Large Linear Systems of Equations arising from Stochastic Modelling*, Proc. PAPM / PROBMIV'02 LNCS 2399, pp.135-151, Springer, 2002.
17. 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.
18. 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.
19. M.Z. Kwiatkowska, D. Parker, H. Qu, *Incremental Quantitative Verification for Markov Decision Processes*, Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 359370, IEEE, 2011.
20. R.W. Vuduc, *Automatic Performance Tuning of Sparse Matrix Kernels*, Ph.D. thesis, University of California, Berkeley, 2003.