Accelerating SpMV Multiplication in Probabilistic Model Checkers using GPUs

Muhammad Hannan Khan[1], Osman Hassan[1], and Shahid Khan[2]

[1] School of Electrical Engineering and Computer Science
National University of Sciences and Technology (NUST), Islamabad, Pakistan
{hkhan.msee17seecs,osman.hasan}@seecs.edu.pk
[2] Software Modeling and Verification, RWTH Aachen University, Aachen, Germany
shahid.khan@cs.rwth-aachen.de

Abstract. Probabilistic model checking is a prominent formal verification technique for analyzing stochastic systems. Probabilistic model checkers hinge upon the sparse matrix-vector (SpMV) multiplications to compute reachability probabilities, i.e., the probability of reaching a target state from a given initial state. Being compute- and memory-intensive task, SpMV is a bottleneck in using probabilistic model checking for analyzing scalable real-world case studies. This paper presents a methodology to accelerate SpMV multiplication in probabilistic model checkers using graphic processing units (GPUs). Since GPUs efficiently execute basic linear algebraic operations such as multiplication, one achieves improvements in computation times. These improvements, however, are not significant in the presence of memory transfer overheads. We apply traditional optimization techniques and hide the memory transfers from the host computer to the GPU inside the state-space-exploration stage. This hiding significantly reduces the latency caused by memory transfers during execution. We implemented the proposed acceleration approach with CUDA-based cuSPARSE API and asynchronous multiple copy algorithms in the probabilistic model checker Storm, with a focus on its SpMV multiplier. In our experiments, we observed 16 times speed up on average over the state-of-the-art.

Keywords: Probabilistic model checking · GPU · Storm · Sparse matrix-vector multiplication

1 Introduction

Model Checking [11] is a widely used formal verification technique [23] that exhaustively builds a behavioral model $\mathcal{M}$ of a given system for a given property ϕ, and automatically verifies if the system exhibits the property $\mathcal{M} \models \phi$. A model checker not only verifies the properties over a model but, in case of a failing property, also provides counterexamples. These counterexamples help developers in understanding and rectifying the non-conforming behavior. As real-world systems pervasively exhibit stochastic behavior, probabilistic model checking

(PMC) is an important extension of model checking [25]. PMC allows verifying stochastic systems, modeled as Markov chains (MCs) or Markov decision processes (MDPs), against probabilistic properties.

Scalability is a persistent issue for both steps of model checking: (1) model building and (2) property verification. The scalability issue in the former step leads to the infamous state-space explosion problem [38]. A promising technique to mitigate the state space explosion problem is the lazy verification approach where partial state space is explored to achieve the results of acceptable precision, see [28]. The scalability issue in the later step is equally important. Internally, the probabilistic model checkers represent the probabilistic behavioral model (state space) as a sparse matrix, and property verification leads to repeated sparse matrix-vector multiplication. As the size of the state-space is translated into the dimensions of the said matrix, the growing size of state space contributes to the complexity of performing arithmetic on such matrices. This increase in complexity results in large computation costs and memory requirements.

Parallel model checking algorithms [3,4,37] traditionally rely on CPU clusters to mitigate the property verification scalability issue, but GPUs have emerged in recent years as the primary compute resource for the application and acceleration of such mathematics. Bosnacki et al. [8] used the Jacobi method in the core sparse matrix and dense vector multiplication to speed up the Markov chain model checking and demonstrated their results on PRISM, a probabilistic model checker, running on GPUs. This was further improved in [9] by enhancing the parallelization of the algorithm, where the memory copying is identified as the main bottleneck for GPU-based algorithms. Cormie-Bowins et al. [12] implemented the matrix multiplication using the Jacobi and the BiCGStab method on GPUs. They compared their work with Bosnacki's advanced GPU-based PRISM. Wijs et al. [40] identified how the wrap-based segments and the modified sparse row (MSR) format matrices improved the sparse matrix-vector multiplication 4.5 times on average. Bylina et al. [10] identified multiple formats for storing sparse matrices to limit the memory footprint and discussed their applicability in GPU-based sparse matrix-vector multiplication. Berger et al. [7] utilized the CUSP library to obtain a significant speed-up when dealing with large models that require multiple iterations to overcome the initial memory copy overhead in STORM [17], a probabilistic model checker. They identified two main challenges: (1) the memory transfer overheads consuming up to 99.96% of time in extreme cases and (2) the lack of hardware support for double-precision floating-point arithmetic. Wijs et al. [39] provided a comprehensive tool, called the GPU-EXPLORE. This tool combines the maximum data inside the 32-bit integers and stores information in the texture memory to mitigate the uncoalesced access overheads. This approach targets only explicit-state model checking; hence, it cannot be utilized for probabilistic models. Bell et al. [5] presented a generic approach to SpMV multiplication on GPUs. They proposed fine-tuned kernels for different storage types of sparse matrices. Moreover, they proposed bypassing the memory latency and computing bottleneck by introducing large-scale GPU-based distributed systems. In all the above-mentioned works, we find latency—

delay due to copying data from a host to device—as the primary bottleneck for GPU-based SpMV multiplication. This underscores the need for a generic multiplication kernel that fully utilizes the available hardware resources.

This paper presents a methodology to further speed up the SpMV multiplications. We investigate the SpMV multiplication in the context of probabilistic reachability probability for discrete-time Markov chains (DTMCs). The methodology (1) leverages upon the traditional optimization methods, (2) hides the memory transfers from the host to the GPU inside the state-space-exploration stage and (3) benefits from the STORM framework-specific algorithms. We use CUDA's native cuSPARSE API and compare the results with the existing CUSP and CPU-based implementations. We also identify some pitfalls and bottlenecks that we encountered when accelerating such algorithms on GPUs. Our selection of STORM is mainly motivated by its promising results [18,24] while maintaining the space for improvement in SpMV multiplication.

2 Preliminaries

2.1 Behavioral Model

Definition 1 (Discrete Time Markov Chains).
A discrete-time Markov chain (DTMC) is a tuple $\mathcal{M} = (S, \mathbf{P}, s_{init}, AP, L)$ [2], where: S is a set of states; $\mathbf{P} : S \times S \to [0,1]$ is the transition probability function such that $\forall\, s : \sum_{s' \in S} \mathbf{P}(s,s') = 1$; $s_{init} \in S$ is an initial distribution; AP is a set of atomic propositions; and $L : S \to 2^{AP}$ is a labeling function.

2.2 Reachability Probability

Reachability probability amounts to computing the probability to reach a predefined set of states $B \subset S$ from any $s \in S \setminus B$. Let $x_s = Pr\{s \models \Diamond B\}$ denote the reachability probability for state s. x_s is computed as:

$$x_{\mathrm{s}} = \underbrace{\sum_{t \in S \setminus B} \mathbf{P}(s,t) \cdot x_{\mathrm{t}}}_{\text{reach } B \text{ via } t \notin B} + \underbrace{\sum_{u \in B} \mathbf{P}(s,u)}_{\text{reach } B \text{ in one step}} \quad . \tag{1}$$

Equation 1 states that either a state $s \in B$ is reached within one step or first a state $t \in S \setminus B$ is reached from which B is reached. If B is not reachable from s, then $x_s = 0$. If $s \in B$, then $x_s = 1$, see [26] for details. For the vector $\mathrm{x} = (\mathrm{x_s})_{\mathrm{s} \in \mathrm{S}}$, where all set of states have a valid path to B, we get from Equation 1

$$x_{\mathrm{s}} = Ax + b, \tag{2}$$

where the matrix A contains the transitional probabilities and b contains the probability of reaching B in one step. Using a (heterogeneous) linear equation system, we rewrite Equation 2 as:

$$(I - A) \cdot x = b, \tag{3}$$

where I is an identity matrix. The probability distribution of $\mathcal{M}$ being in a state after n transitions, given that the computation starts with an initial state vector s_{init}, is denoted by $\theta_{\mathrm{n}}^{\mathcal{M}}$ and computed as

$$\theta_{\mathrm{n}}^{\mathcal{M}} = \mathbf{P} \cdot \mathbf{P} \cdot .. \cdot \mathbf{P} \cdot s_{\mathrm{init}} = \mathbf{P}^{\mathrm{n}} \cdot s_{\mathrm{init}} \tag{4}$$

Since calculating the n power of the matrix is a computationally expensive operation [2], $\theta_{\mathrm{n}}^{\mathcal{M}}$ is calculated by recursive matrix-vector multiplications.

2.3 Sparse-Matrix representations

The general sparse matrix-vector multiplication equation is $y = \alpha\mathbf{A}x + \beta\mathbf{y}$, where: $\mathbf{A}$ is the sparse matrix of size $Cols \times Rows$; x (y) is a dense vector of size $Cols$ ($Rows$); and both α and β are scalars. The process of multiplication can, therefore, be summarized by the following equation

$$y_i = \sum_{A_{i,j} \neq 0} A_{i,j} \cdot x_j + \beta \cdot y_i \tag{5}$$

Equation 5 indicates that the operation to be performed at each Non-zero value (V_{NNZ}) of the sparse-matrix A results in an overall $(V_{NNZ} + Rows) \cdot 2$ floating point operations.

This paper utilizes the compressed sparse row (CSR) format due to (1) its wide utilization in STORM and (2) its ability to provide a balanced computation across matrices of different sizes and sparsity, as identified in [10, 21]. The CSR format divides the matrix into 3 arrays: (1) non-zero data values, (2) their column indices, and (3) offsets of each row represented in the data.

2.4 GPU Programming

Graphical processing units (GPUs) are highly parallel programmable processors. They specialize in accelerating the low-level algorithms, which have large computational requirements and are parallelizable. GPUs follow the single instruction multiple data (SIMD) programming model, i.e., the GPU processes multiple data elements in parallel using the same instruction.

Next, we describe two of the most widely used GPU accelerators and their pros and cons. Both have their unique programming models and provide multiple tools to allow optimizations and computations of algorithms.

OpenCL. The OpenCL [32] provides a cross-platform environment that can be easily ported to multiple architectures like the CPUs, GPUs, digital signal processors (DSPs), and even field-programmable gate arrays (FPGAs). Unfortunately, due to this heterogeneous behavior, OpenCL is not device-specific; hence it does not specialize in any particular hardware.

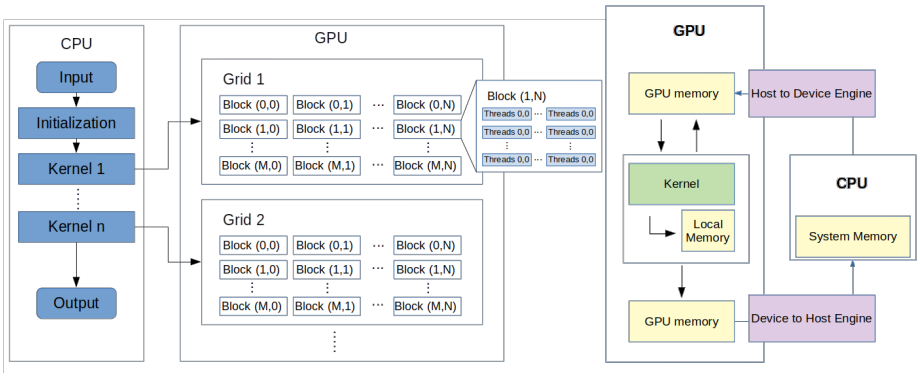
CUDA. To cater for general-purpose computing on GPUs (GPGPU), NVIDIA has developed the compute unified device architecture (CUDA) [36]. As the scientific community is the primary user of this model, we have many off-the-shelf APIs available that target most of the complex and commonly used tasks.

A survey of existing work considers CUDA as a better option due to the availability of specialized APIs, like the CUBLAS and the cuSPARSE [13, 34]. Our use of CUDA is mainly motivated by the following two reasons:

- Fang [19] shows that CUDA significantly outperforms OpenCL in arithmetic computations but lacks in data movement,
- CUDA provides us with multiple open-source tools that reduce the time and complexity of converting the existing code to the CUDA platform.

The GPU devices are typically mounted on the peripheral component interconnect express (PCIe) socket when connected as a co-processor. These devices act as slaves while relying mostly on the CPU to give instructions in a form of tiny applications called kernels. Fig. 1a depicts this behavior of sequential executions in a CUDA application. All data, which needs to be processed, must be transferred to the GPU via the PCIe interface, as shown in Fig. 1b.

The CUDA programming model provides the user with an abstraction of this parallel architecture in the form of directives governing the systems ability to call the SIMD instructions, memory movement and thread synchronizations. At the lowest level of a CUDA subsystem, we have a simple thread that performs the task assigned by the kernel. A thread block is a batch of threads that share memory and perform a task either collectively or individually. The threads of multiple blocks do not co-operate and require extensive synchronizations. Fig. 1a shows how various blocks can be combined to form a one, two, or three dimensional units called grids.



(a) CPU controlled GPU (b) Mem. transfer over PCIe

Fig. 1: GPU programming model

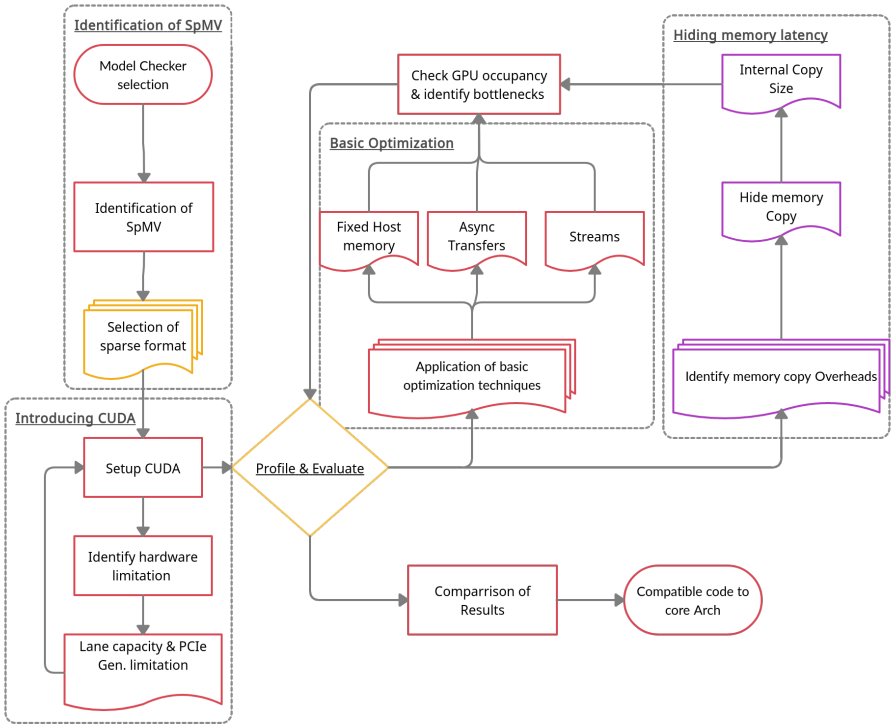


Fig. 2: Proposed methodology diagram

3 Proposed Optimization Flow

This section outlines a strategy, based on conventional and non-conventional methods, to reduce the compute requirements of a probabilistic model checker. The overall approach is to determine a set of pre-requisites rules along with system-specific techniques to optimize the SpMV multiplication, see Fig. 2.

Typically, in optimization problems, the whole system is kept intact and only the problem set is cherry-picked for acceleration. Likewise, we identify possible injection points in the system where we introduce the custom code. We introduce Algorithm 1 that describes the proposed optimization strategy along with key points of code injection.

3.1 Identification of the SpMV

As discussed in Section 2, the process of calculating the probabilities using the matrix-vector multiplication involves very high computation requirements. In the proposed approach, we identify algorithms that first build the model in terms of the sparse-matrix and then perform model checking. The approach is in line

with the STORM's *sparse* and *hybrid* engine. This approach allows us to shift the relatively sequential operation of state exploration on the CPU along with a sparse matrix solver on the GPU. As the explored state of the model is sparse, we use different sparse storage formats to further reduce the memory footprint.

3.2 Introducing CUDA

We introduce a bottom-up technique where we first target and isolate the STORM multiplier and replace it with CUDA based cuSPARSE API. CUSP [6] is another open-source CUDA library that performs the SpMV multiplication. Our choice of using cuSPARSE over CUSP is primarily based on the fact that cuSPARSE has coalesced global memory accesses and provides better occupancy of GPUs. Moreover, it allows asynchronous executions with respect to the host and may return control back to the user before completion. Another reason is that, unlike CUSP, cuSPARSE has been integrated into the CUDA toolkit. Thanks to this integration, cuSPARSE is regularly updated and actively maintained with the support for the state-of-the-art drivers, technology and CUDA enhancements, such as using tensor cores [31], NVIDIA tool extension (NVTX), etc. We discuss, in the experimental results section, how cuSPARSE was observed to be faster than the CUSP implementation.

In Algorithm 1-Phase C, we create a multiplier that accepts the sparse-matrix A_{CSR} in the CSR format, initial vector x_{init} and N as the number of iterations of the multiplier. The copying of data to and from the GPU is required before processing any data. This is followed by calling the cuSPARSE's single or double precision SpMV function. Initially, we need to calculate the memory required for the GPU using the equation:

$$M_{Req} = (\mathcal{P}_{Val} \cdot (||V_{NNZ}|| + ||x_{init}|| \cdot 2) + (\mathcal{P}_{Ind} \cdot (||Offset_{Row}|| + ||V_{NNZ}||))) \quad (6)$$

where $\mathcal{P}_{Val}$ is the precision of $\mathrm{V_{NNZ}}$, $Offset_{Row}$ is the row offset vector of A_{CSR} and V_{NNZ} is the vector of values in A_{CSR}. The M_{Req} is the maximum memory that the GPU can accommodate without dividing the matrix into a subset to compute the SpMV multiplication.

Another limiting factor is the PCIe interface of the GPUs. Table 1 presents an overview of different PCIe interfaces available and their theoretical performance.

Table 1: PCIe types and data rates [1, 20]

Interface	Data-rate supported
PCIe x1 Gen 3	1 GB/s
PCIe x8 Gen 3	8 GB/s
PCIe x16 Gen 3	16 GB/s
PCIe x16 Gen 4	32 GB/s

Memory copy is the main source of latency in all GPU applications that are generated as a consequence of the type of PCIe slot selected. Fig. 4 illustrates

the ratio of the time spent on kernel compute vs memory transfers. We observed that in the mobile versions of GPUs, the x8 PCIe interface is commonly used due to the limited availability of space, and on the other hand, fast x16 interfaces based on Gen 3, or more recently Gen 4, are used in desktop computers.

Algorithm 1 Complete optimized algorithm

Phase-A

```
1: procedure BUILDING SPARSE MATRIX(M)
2:     ...
3:     procedure STATE EXPLORATION(s, Cp_min)
4:         ToExplore = {s_0}
5:         Discovered = { }
6:         s_old = { }
7:         while ToExplore ≠ Null do
8:             s_picked ∈ ToExplore
9:             s_x = FindSuccessors(s_picked)
10:            Discovered = Discovered ∪ s_picked
11:            ToExplore = (ToExplore \ s_picked) ∪ s_x
12:            if s_x is ≥ Cp_min then                    ▷ 1st insertion of CUDA
13:                Copy-Asynchronous(V_Chunks(i)) = s_x \ s_old
14:                s_old = s_x
15:                Cp_min = Cp_min*1.75
16:            end if
17:        end while
18:    end procedure
19:    ...
20: end procedure
```

Phase B

```
21: procedure REASSEMBLE AND COPY(V_Chunks, x_init)    ▷ 2nd insertion of CUDA
22:     Copy-Asynchronous(x_init )_Host→Device
23:     for each item i in V_Chunks do
24:         Copy-Optimized(V_Chunks(i)→A_CSR)_Device→Device
25:     end for
26: end procedure
```

Phase C

```
27: procedure CUDA MULTIPLIER(A_CSR, x_init, N)        ▷ 3rd insertion of CUDA
28:     for values of N do
29:         y = cuSPARSE SpMV(A_CSR · x_init )
30:         Swap pointers(x_init←y)
31:     end for
32: end procedure
```

3.3 Basic Optimizations

Several possible optimizations can be considered, and at various levels, ranging from overlapping data transfers with computation down to fine-tuning floating-

point operation sequences. NVIDIA provides a best practice guide [14] that outlines all traditional optimization strategies. In our context, minimizing the transfer of data between the host and the device is essential. This minimization might lead to sacrificing the computations on GPU to run kernels that otherwise exhibit similar performance on the host CPU.

Pinned Memory *Page-locked* or *pinned memory* transfers attain the highest bandwidth between the host and the device. Since the GPU is not able to access the data directly from the *pageable* host memory, the CUDA driver must first allocate a temporary page-locked, or "pinned" memory. The required data is first copied to the pinned array and then transferred from the pinned array to the device memory. On PCIe $x16$ Gen3 cards, for example, the pinned memory can attain transfer rates of about 12 GB/s.

Async-Transfer *CudaMemcpy* provides the basic data transfer between the host and the device by blocking the execution control of the host thread until the data transfer is complete. The *asynchronous data transfer* function, *cudaMemcpyAsync*, is a non-blocking variant of the *cudaMemcpy* in which the control is returned immediately to the host thread. This allows the user to queue multiple copy commands to constantly engage the *Copy-engine* while the CPU is free to perform other tasks. Utilizing *async-transfers* along with *pinned memory* techniques, forms the "Copy-Asynchronous" procedure in Algorithm 1.

Stream is a pipeline within the CUDA API that allows a sequence of operations to be executed on the device in a given order, defined host-side. These streams, while maintaining the order within the context they run, allow the execution of multiple streams that can be interleaved or executed concurrently. Fig. 3 shows how multiple streams can fully utilize hardware that remains unused when a sequential flow is implemented.

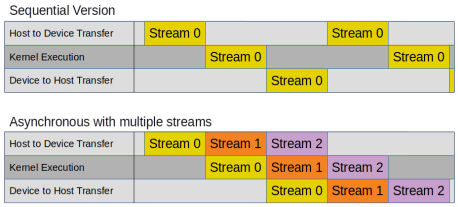


Fig. 3: Stream allowing sequential operation into concurrent operations

Some other optimization techniques include batching small transfers together in order to fully utilize the PCIe transfer speeds and handling of strided accesses using coalesced reads from the global memory.

3.4 Hiding Memory Latency

While traditional optimization techniques provide multi-fold speed-ups, most algorithms require custom optimizations that exploit the flow of the system to create room for specific code insertion. It can be observed form Fig. 4 that the bulk of memory latency comes from the copying of the three CSR vectors replacing the transitional matrix. Therefore, minimizing the time taken for this operation is of utmost importance.

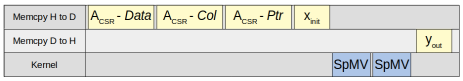


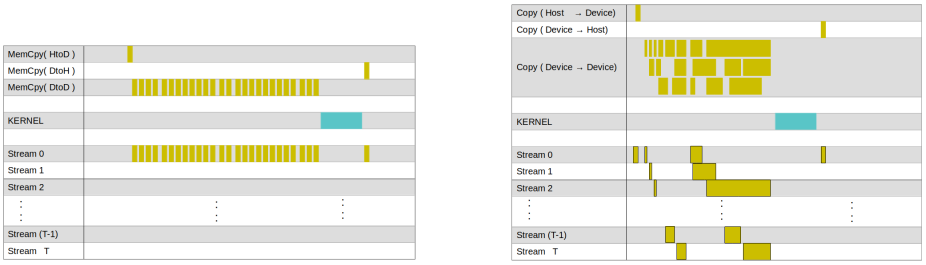
Fig. 4: Memory latency generated when computing SpMV multiplication.

For repeated $Y_i = Ax_i + b$, the value of x_i is equal to Y_{i-1}, this adds latency due to the transfer of vector from device to device. In cases where two variables of equal length within the CUDA environment needs to be copied, we swap their memory pointers followed by reassigning the context through the cuSPARSE API. Algorithm 1-C follows this logic in the "Swap pointers" procedure to remove the copy overhead between x_{init} and y.

STORM uses its sparse-matrix builder utility to create the transitional matrix $\mathbf{A}$ by state exploration for each successor state (s_x) and assigns it as a row of the matrix. Since these rows, depicting state transitions, are fixed, we propose Algorithm 1-A as an extension of the state exploration process. We introduce the variable $V_{Chunks}(i)$ where each i is a pointer to memory containing rows of matrix $\mathbf{A}$. Since all memory copying is asynchronous, the control is handed back to the API as soon as the command is executed, ensuring that the delay in the state exploration is minimal. Algorithm 1-B shows how we efficiently rearrange the pointers inside a contiguous memory, once all data is copied inside the memory at the multiplication point.

Recalling Fig. 1a, all instructions to the GPU are directed by the CPU. When the number of states and choices increase, the number of copy pointers and the instructions required to rearrange them also increases, as shown in Fig. 5a. This sequential execution is catered by utilizing the maximum streams, since there is no limitation on the parallel copy operation internally in the device.

To optimize the copying speeds over the PCIe, it is typically recommended to have large copy sizes [33] by batching small transfers together. We introduced a limit Cp_{min} in Algorithm 1-A to ensure that the copy process groups multiple rows of matrix $\mathbf{A}$ together. The number of rows is increased at every copy to ensure that the algorithm adapts as the state-space increases to accommodate the size of the end transitional matrix. Fig. 5b shows a significant performance upgrade after the application of both of the optimization strategies.



(a) Sequential reassemble and copy

(b) Reassemble and copy with multiple streams and batched data

Fig. 5: Illustration of Nvidia's Visual Profiler output showing memory copy and SpMV compute

3.5 Profile and Evaluate

Profiling and evaluation is an essential step after each optimization strategy is implemented. We use the NVIDIA Visual Profiler [16] for GPUs with compute capability of less than 8.0 and NVIDIA compute [15] for compute capability greater than or equal to 8.0. Another important factor in evaluating the quality of optimization is the occupancy of the GPU, i.e., the amount of processor usage by the hardware [22]. The quality of an optimization is proportional to the occupancy per streaming multiprocessors (SMs).

For our experiments, we selected the same test vectors that were previously used to evaluate the PRISM model checker [29]. We focus on the NAND and Herman case studies because they provide a wide range of test vectors with varying size.

4 Experimental Evaluation

The standard STORM multiplier $y = Multiplier(A, x_{init}, N)$ takes three parameters as input: (1) a sparse-matrix A, (2) a dense initial state vector x_{init} and (3) number of times to perform multiplication N. The multiplier returns a dense vector y depicting step-bounded readability probability of each state. For the implementation of the SpMV, we randomized the dense vectors in the range $[0, 1]$ and increased the value of N.

We introduce the term "*Complete CUDA*" to collectively represent the basic optimization techniques and hiding the memory transfers inside the state-space exploration. The experiments are performed on three different combinations of CPUs and GPUs each scaling in performance to accommodate technological advances within each generation. The combinations are as follows:

1. Intel i7-7700 CPU @ 2.8GHz with GTX 1050 PCIe 3rd Gen x8 lane;
2. Intel i7-6700 CPU @ 3.4GHz with GTX 1080 PCIe 3rd Gen x16 lane;

3. AMD RYZEN 3970x 32-core @ 3.7GHz with RTX 3090 PCIe 4th Gen x16 lane.

We present the results of combination 2 here, and more details for 1 and 3 can be found in [27].

All systems run the standard Ubuntu 18.04 LTS with CUDA toolkit 11.2. We compare our SpMV implementation with the Storm built-in multiplier function, and the cuSPARSE implementation with the CUSP implementation. All tests are conducted assuming a double precision with the multiplier count = 2, making the worst-case scenario for GPUs due to such models' requirements for low latency. We provide the results for each benchmark as the value of N is increased. Finally, we illustrate the difference between the cuSPARSE and CUSP.

4.1 NAND Case Study

This case study [35] concerns NAND multiplexing, which is a technique for ensuring reliable computation using unreliable devices. There are two variables that change the dynamics of the model: (1) N is the number of inputs in each bundle and (2) K is the number of restorative stages. The experimental results are depicted in Table 2 and Fig. 6a.

Table 2: NAND: comparing optimizations with Storm multiplier

NAND constants		**Size of Matrix**	**Sparsity**	**Basic** CUDA	**Complete** CUDA	**Storm**	**Speed-up** factor
N	K		%	mS	mS	mS	$\frac{\text{Storm}}{Complete}$
20	1	78332 x 78332	99.99802	0.382	0.27	3.1045	11.5
20	2	154942 x 154942	99.999001	0.69	0.424	4.019	10
20	3	231552 x 231552	99.99933	0.92	0.514	6.208	12
20	4	308162 x 308162	99.9995	1.21	0.69	8.2	11.9
40	1	1004862 x 1004862	99.99984	3.61	1.769	30.594	17.3
40	2	2003082 x 2003082	99.99992	6.91	3.391	54.716	16.1
40	3	3001302 x 3001302	99.99995	10.3	5.002	82.158	16.4
40	4	3999522 x 3999522	99.99996	13.68	6.66	112.76	17
60	1	4717592 x 4717592	99.99997	16.48	7.861	131.434	16.7
60	2	9420422 x 9420422	99.999983	43.33	15.614	265.969	17
60	3	14123252 x 14123252	99.999989	64.581	23.296	392.483	16.9
60	4	18826082 x 18826082	99.999992	79.22	31.142	528.142	17

The measurements against the Storm implementation show speed-up of 16 times on average with the *Complete CUDA* GPU implementation. In smaller matrices, we observe the basic optimization strategy giving similar results as the complete optimizations. This is expected since the latency is less significant with smaller matrices.

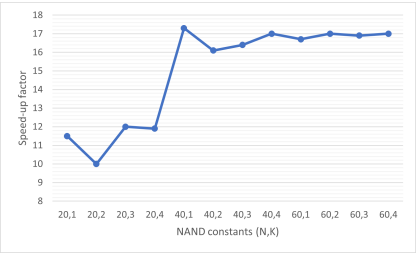
Table 3: Herman: comparing optimizations with STORM multiplier

Model Name	**Matrix Size**	**Sparsity**	**Basic** CUDA	**Complete** CUDA	**Storm**	**Speed-up** factor
		%	mS	mS	mS	$\frac{\text{STORM}}{Complete}$
Herman3	8 x 8	56.25	0.2	0.202	0.0015	0.007
Herman5	32 x 32	76.17	0.2	0.217	0.0025	0.012
Herman7	128 x 128	86.64	0.21	0.197	0.0070	0.036
Herman9	512 x 512	92.5	0.21	0.209	0.525	2.513
Herman11	2048 x 2048	95.8	0.31	0.242	4.624	19.11
Herman13	8192 x 8192	97.62	1.83	0.603	16.026	26.58
Herman15	32768 x 32768	98.66	15.19	3.573	125.495	35.12

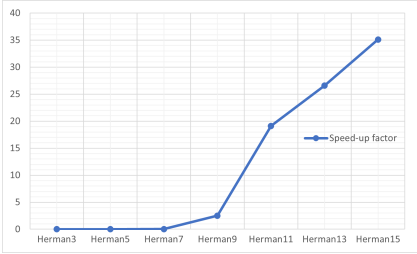
4.2 Herman Case Study

The self-stabilizing algorithm Herman [30] operates synchronously in an oriented ring, where the communication is unidirectional in the ring. In this protocol, the number of processes in the ring must be odd. Our choice of Herman stems from the fact that it exhibits lower sparsity in comparison to other benchmark models and thus leads to a faster multiplication but this speed-up is compromised due to an increase in the memory copy operations from the host to the device. We also evaluate how using GPUs to solve small matrix-vector multiplication is counter-intuitive since the setup cost of matrix multiplication is greater than the complete multiplication on CPU. Execution times can be seen in Table 3 and comparisons are illustrated in Fig.6b.

For the Herman model, the proposed approach initially performs worse than the original STORM multiplier. We observe a minimum time of 0.2 ms for all matrices and since the STORM multiplier can handle matrices of up to 512×512 in under 0.5 ms, it significantly decreases the speed-up factor but with models greater than 2048×2048 we see an up to 35 times increase in performance in the proposed approach.



(a) Speed-up factor in NAND model.



(b) Speed-up factor in Herman model.

Fig. 6: Results from NAND and Herman model.

Unlike the NAND model, which on average saw twice the performance gain when comparing the basic with *Complete CUDA* optimizations, as shown in Fig. 7a, we see a higher difference in favor of complete optimizations in the Herman model. This is due to lower sparsity, which creates a higher cost for memory copying if the transfer time is not included in the state-space exploration.

4.3 Increasing Value of N

All of the above observations have been made assuming a value of $N = 2$ denoting that the matrix-vector multiplication is performed twice before termination. With an increased value of N, the cost of performing SpMV multiplication once can be computed as

$$Ratio_n = \frac{Time_{N=1}}{Time_{N=n}} \cdot n \tag{7}$$

with n being the multiplication count. We observe from Table 4, that the time taken for a single SpMV multiplication instance reduces with an increased value of N in all models. This behavior, as illustrated in Fig. 7b, is expected since GPUs traditionally use such tactics to compensate for the initial latency caused by memory transfers [22].

Table 4: Execution time of SpMV multiplication for each value of N

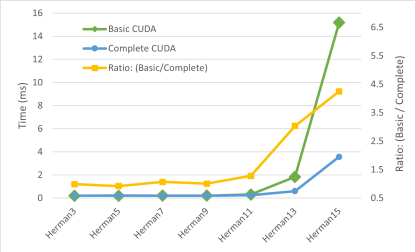
	Multiplication Count - N					
Name	**1**	**4**	**6**	**10**	**50**	**100**
Herman3	183	56.75	40.833	28.7	15.46	13.78
Herman5	180	55.5	41.667	29.9	15.96	14.24
Herman7	184	61	41.67	29.5	16.12	13.73
Herman9	188	62.5	40.167	29.2	15.66	13.84
Herman11	214	69	54.5	41.8	26.12	24.52
Herman13	467	207.5	177.3	155	127.34	122.92
Herman15	2633	1361	1219.67	1107.4	968.86	951.78

4.4 CUSP vs cuSPARSE

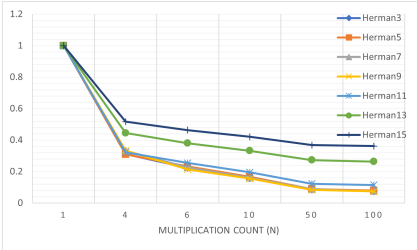
The CUSP vs cuSPARSE kernel comparison is performed for different matrices and Table 5 shows a steady lag that CUSP maintains behind the cuSPARSE library. We found that the resource utilization per streaming multiprocessor in the cuSPARSE API resulted in a lower time to compute a kernel of the same dimension and value as compared to the CUSP implementation.

4.5 Comparing GPUs

We also compared hardware on the basis of generation with the RTX-3090 being the top-of-the-line GPU using PCIe 4.0 x16 lanes followed by the GTX-1080



(a) Basic and complete optimizations with the ratio of Basic/complete.



(b) $Ratio_n$ of each model with increased value of N.

Fig. 7: Comparison results of optimizations and increased N.

Table 5: Kernel duration for the Herman model over CUSP and cuSPARSE

Model	**CUSP time**	**cuSPARSE time**	$\frac{CUSP}{cuSPARSE}$
Herman3	18.23	10.312	1.768
Herman5	24.29	10.27	2.365
Herman7	32.56	9.69	3.36
Herman9	107.2	10.03	10.69
Herman11	437.7	22.65	19.32
Herman13	2836	123.44	22.98
Herman15	19720	956.78	20.61

and GTX-1050 (x8 lane). From Table 6, we find that the RTX-3090 performs up to 228% faster on matrices that require a higher memory bandwidth, similar to those of the Herman model, while on high sparsity matrices we see an improvement of 137.5%.

For our results, we compare the output probabilities with the ones obtained via Storm's multiplier and found both of them to be identical. The results identify that for small matrices, the GPU implementation is not recommended since the time taken for such SpMV multiplications on CPU was observed to be less than 100 microseconds. On the other hand, we see a significant performance gain of up to 80 times on high-end GPUs with models that have large matrices and high transitions per state, and up to 20 times on average in highly sparse matrices. Furthermore, when applying the multiplication on the sparse-matrices with only basic optimization techniques, we observe that, on average, 83% of the time is spent on memory transfers while this ratio reduces to 65% when memory copy latency is hidden inside the state-space exploration. This is due to CUDA's ability to allow multiple fast streams when copying data within the device as compared to copies from the host. Finally, we see that older GPU generations also provide speed-ups of up to 8 times in comparison to the Storm multiplier on CPU.

Table 6: Speed-up factor over different generations on Nand and Herman model

Model	**RTX-3090** (Gen.4 - 16 Lane)	**GTX-1080** (Gen.3 - 16 Lane)	**GTX-1050** (Gen.3 - 8 Lane)
NAND - 20,1	4.905	11.5	5.344
NAND - 20,2	9.871	10	6.205
NAND - 20,3	8.742	12	6.589
NAND - 20,4	15.75	11.9	7.059
NAND - 40,1	15.16	17.3	8.443
NAND - 40,2	17.56	16.1	8.352
NAND - 40,3	19.13	16.4	8.006
NAND - 40,4	19.16	17	8.090
NAND - 60,1	19.77	16.7	7.926
NAND - 60,2	21.60	17	7.846
NAND - 60,3	22.11	16.9	7.927
NAND - 60,4	22.63	17	8.249
Herman3	0.031	0.007	0.05
Herman5	0.061	0.012	0.108
Herman7	0.143	0.036	0.34
Herman9	0.73	2.513	2.67
Herman11	7.21	19.11	12.2
Herman13	37.49	26.58	16.08
Herman15	80.31	35.12	18.59

5 Conclusion

This paper has presented a GPU-based methodology to optimize sparse-matrix vector multiplications for probabilistic model checking. Significant improvements in performance are achieved by enabling optimizations on the memory transfer step and by using built-in CUDA APIs. Several aspects of the proposed approach are studied. Experiments revealed a speed up of 16 times over the state-of-the-art.

All GPU assisted applications are limited by their global memory utilization. As state-of-the-art hardware crams maximum 80 Gigabytes of memory, the next step towards the GPU aided model checkers will be to cater for matrix-vector multiplications where the size of the variables exceed the limit of the GPU memory. Abstraction techniques to reduce the size of model are generally applied to merge multiple states with indistinguishable behaviour. Techniques such as bisimulation minimization could either be applied in the GPU or output of the CPU-based implementation can be imported and merged in the GPU memory. Extension to a more generic problem set, such as nested bounded probabilistic model checking along with cross-platform comparison with other hardware accelerators and implementation of simulation algorithms such as statistical model checking can be explored as possible future avenues. Another interesting future direction will be to implement the state-space exploration inside the GPU. Since this pre-processing step takes a significant amount of time, GPU-based exploration can introduce a parallel implementation to find successor states. This approach will also avoid repeated memory movement between the host and the GPU; thus it will inherently preempt the primary latency factor.

References

1. Ajanovic, J.: Pci express 3.0 overview. In: Proceedings of Hot Chip: A Symposium on High Performance Chips. vol. 69, p. 143 (2009)
2. Baier, C., Katoen, J.P.: Principles of model checking. MIT press (2008)
3. Barnat, J., Bloemen, V., Duret-Lutz, A., Laarman, A., Petrucci, L., van de Pol, J., Renault, E.: Parallel model checking algorithms for linear-time temporal logic. In: Handbook of Parallel Constraint Reasoning, pp. 457–507. Springer (2018)
4. Barnat, J., Brim, L., Stříbrná, J.: Distributed LTL model-checking in SPIN. In: International SPIN Workshop on Model Checking of Software. pp. 200–216. Springer (2001)
5. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on CUDA. Tech. rep., Citeseer (2008)
6. Bell, N., Garland, M.: Cusp: Generic parallel algorithms for sparse matrix and graph computations. Version 0.3. 0 **35** (2012)
7. Berger, P.: GPU-aided model checking of markov decision processes (2014)
8. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: GPU-PRISM: An extension of prism for general purpose graphics processing units. In: 2010 Ninth International Workshop on Parallel and Distributed Methods in Verification, and Second International Workshop on High Performance Computational Systems Biology. pp. 17–19. IEEE (2010)
9. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: Parallel probabilistic model checking on general purpose graphics processors. International Journal on Software Tools for Technology Transfer **13**(1), 21–35 (2011)
10. Bylina, B., Bylina, J., Karwacki, M.: Computational aspects of GPU-accelerated sparse matrix-vector multiplication for solving markov models. Theoretical and Applied Informatics **23**, 127–145 (2011)
11. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM transactions on Programming Languages and Systems (TOPLAS) **16**(5), 1512–1542 (1994)
12. Cormie-Bowins, E.: A comparison of sequential and GPU implementations of iterative methods to compute reachability probabilities. arXiv preprint arXiv:1210.6412 (2012)
13. Corporation, N.: The api reference guide for cusparse (2021), https://docs.nvidia.com/cuda/cusparse/index.html
14. Corporation, N.: Cuda c++ best practices guide (2021), https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html
15. Corporation, N.: Nvidia nsight compute (2021), https://developer.nvidia.com/nsight-compute
16. Corporation, N.: Nvidia visual profiler (2021), https://developer.nvidia.com/nvidia-visual-profiler
17. Dehnert, C., Junges, S., Katoen, J.P., Volk, M.: A storm is coming: A modern probabilistic model checker. In: International Conference on Computer Aided Verification. pp. 592–600. Springer (2017)
18. Fabarisov, T., Yusupova, N., Ding, K., Morozov, A., Janschek, K.: The efficiency comparison of the Prism and storm probabilistic model checkers for error propagation analysis tasks. Industry 4.0 **3**(5), 229–231 (2018)
19. Fang, J., Varbanescu, A.L., Sips, H.: A comprehensive performance comparison of CUDA and OpenCL. In: 2011 International Conference on Parallel Processing. pp. 216–225. IEEE (2011)

20. Gonzales, D.: Pci express 4.0 electrical previews. In: PCI-SIG Developers Conference (2015)
21. Greathouse, J.L., Daga, M.: Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In: SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 769–780. IEEE (2014)
22. Harris, M.: Optimizing cuda. SC07: High Performance Computing With CUDA **60** (2007)
23. Hasan, O., Tahar, S.: Formal verification methods. In: Encyclopedia of Information Science and Technology, Third Edition, pp. 7162–7170. IGI Global (2015)
24. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker Storm. arXiv preprint arXiv:2002.07080 (2020)
25. Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate probabilistic model checking. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 73–84. Springer (2004)
26. Katoen, J.: The probabilistic model checking landscape. In: LICS. pp. 31–45. ACM (2016)
27. khan, H.: Storm-cuda (2021), https://github.com/khan-hannan/StoRM-CUDA
28. Khan, S., Katoen, J., Volk, M., Bouissou, M.: Scalable reliability analysis by lazy verification. In: NFM. Lecture Notes in Computer Science, vol. 12673, pp. 180–197. Springer (2021)
29. Kwiatkowska, M., Norman, G., Parker, D.: The PRISM benchmark suite. In: 9th International Conference on Quantitative Evaluation of SysTems. pp. 203–204. IEEE CS press (2012)
30. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic verification of herman's self-stabilisation algorithm. Formal Aspects of Computing **24**(4), 661–670 (2012)
31. Markidis, S., Der Chien, S.W., Laure, E., Peng, I.B., Vetter, J.S.: Nvidia tensor core programmability, performance & precision. In: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 522–531. IEEE (2018)
32. Munshi, A., Gaster, B., Mattson, T.G., Ginsburg, D.: OpenCL programming guide. Pearson Education (2011)
33. Nambiar, P.P., Saveetha, V., Sophia, S., Sowbarnika, V.A.: GPU acceleration using CUDA framework. International Journal of Innovative Research in Computer and Communication Engineering **2**(3), 200–205 (2014)
34. Naumov, M., Chien, L., Vandermersch, P., Kapasi, U.: Cusparse library. In: GPU Technology Conference (2010)
35. Norman, G., Parker, D., Kwiatkowska, M., Shukla, S.: Evaluating the reliability of NAND multiplexing with PRISM. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **24**(10), 1629–1637 (2005)
36. Sanders, J., Kandrot, E.: CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional (2010)
37. Stern, U., Dill, D.L.: Parallelizing the murϕ verifier. In: International Conference on Computer Aided Verification. pp. 256–267. Springer (1997)
38. Valmari, A.: The state explosion problem. In: Advanced Course on Petri Nets. pp. 429–528. Springer (1996)
39. Wijs, A., Neele, T., Bošnački, D.: GPUexplore 2.0: unleashing GPU explicit-state model checking. In: International Symposium on Formal Methods. pp. 694–701. Springer (2016)

40. Wijs, A.J., Bošnački, D.: Improving GPU sparse matrix-vector multiplication for probabilistic model checking. In: International SPIN Workshop on Model Checking of Software. pp. 98–116. Springer (2012)