GPU-PRISM: An extension of PRISM for General Purpose Graphics Processing Units

(Tool Paper)

Dragan Bošnački*, Stefan Edelkamp†, Damian Sulewski†, and Anton Wijs*
*Eindhoven University of Technology, The Netherlands
†TZI, Universität Bremen, Germany

***Abstract*—We present an extension of the model checker PRISM for (general purpose) graphics processing units (GPUs). The extension is based on parallel algorithms for probabilistic model checking which are tuned for GPUs. In particular, we parallelize the parts of the algorithms that boil down to linear algebraic operations, like solving systems of linear equations and matrix vector multiplication. These computations are performed very efficiently on GPGPUs which results in considerable runtime improvements compared to the standard versions of PRISM. We evaluated the extension of PRISM on several case studies in which we observed significant speedup over the standard CPU implementation of the tool.**

***Keywords*-probabilistic model checking; model checker PRISM; parallel algorithms; GPU;**

I. Introduction

We present an extension of the probabilistic model checker PRISM [4] which exploits the computation power of (general purpose) graphics processing units (GPUs). The current implementation of the tool is based on the CUDA architecture for NVIDIA graphics cards [3], however the framework is rather general and it can be adapted seamlessly to other graphics cards, as well as other model checking tools. GPU-PRISM is fully compatible with standard PRISM.

Probabilistic Model Checking: Probabilistic model checking [5] is a branch of model checking which has been successfully used for the analysis of models that have a probabilistic/stochastic nature. These models cover a broad spectrum of applications ranging from communication protocols like FireWire and Bluetooth, to biological networks that model gene expression.

In traditional model checking one usually aims at proving absolute logical correctness of the analyzed model against a given property. In probabilistic model checking the correctness of the properties is quantified with some probability. The properties are expressed in extensions of the traditional temporal logics such that the quantitative probabilistic aspects are captured.

The Probabilistic Model Checker PRISM: PRISM [4] is a probabilistic model checker which was developed initially at the University of Birmingham and currently is being developed at the University of Oxford. PRISM is an open-source tool and written in Java and C++. During the years the tool has gained a significant popularity and it has been tested on various case studies. A quite comprehensive summary of PRISM applications can be found on the tool web page `http://www.prismmodelchecker.org`.

PRISM supports three types of models: discrete- and continuous Markov chains (DTMCs and CTMCs), and Markov decision processes (MDPs), The models are specified using the PRISM modeling language which is based on the Reactive Modules formalism. Systems are described as a set of modules executed in parallel. Each module contains transitions to which probabilities are associated in various ways, depending on the model type.

Properties are specified in the logics PCTL and CSL, which are probabilistic extensions of the logic CTL. PCTL is used to specify properties of DTMC models, whereas CSL is used in the context of CTMCs.

Parallel Probabilistic Model Checking on GPUs: The main difference between traditional and probabilistic model checking is that the latter has a numerical component to capture the probabilities [5]. This numerical part hinges critically on linear algebraic operations like matrix-vector multiplication and scalar product of two vectors.

General purpose graphics processing units (GPUs) are powerful coprocessors that outgrew their applications in graphics. Since linear algebraic operations can be implemented very efficiently on GPUs significant speedups can be achieved compared to the sequential counterparts of the probabilistic model checking algorithms. Motivated along this line of reasoning, in a previous work [1] we introduced parallel probabilistic model checking on GPUs.

Previous parallel algorithms were exclusively designed for distributed architectures, i.e., computer clusters. The main difference compared to GPUs is that by using fast shared memory one can avoid the costly communication overhead between the parallel processors. Besides that, in GPU programming one should also take into account the various types of memories since the performance of the implementation depends significantly on the memory latencies.

As already mentioned, GPU-PRISM is based on the CUDA architecture. Compute Unified Device Architecture (CUDA) is an interface by the manufacturer NVIDIA [3] which facilitates the use of GPUs beyond graphics oriented applications. CUDA programs are basically extended C

programs which comprise features like: special declarations to explicitly place variables in some of the memories (e.g., shared, global, local), predefined keywords (variables) containing the block and thread IDs, synchronization statements for cooperation between threads, runtime API for memory management (allocation, deallocation), and statements to launch functions on GPU.

Related Work and Novel Elements: Compared to the above mentioned precursor work [1], we parallelize the algorithms to a greater extent. In particular, for each of the three types of models supported by PRISM we parallelize the algorithms for bounded until. By running a series of matrix-vector multiplications, the critical parts of these algorithms, on GPUs we achieve significant runtime improvements.

II. Implementation

The implementation of the extension was done in a modular fashion, since it required modification of just several files in the subfolder `sparse` of the standard PRISM distribution. For the parallelization of the Jacobi algorithm which solves systems of linear equations we replaced the corresponding parts of the code with methods using functions to be run on the GPU cores.

We implemented two different approaches to parallelizing the multiplication of an $n \times n$ matrix A with an n vector x, resulting in an n vector b. In one approach (M_1), for the computation of each entry in b, one core is used, i.e. core i multiplies each non-zero entry in row i of A with the i-th element in x, and computes the sum of these multiplications. In the other approach (M_2), first $n \times n$ cores are used to perform the direct multiplications, i.e. each core only performs one multiplication of a specific entry in A with the corresponding entry in x. After that a *backwards inclusive segmented scan* is performed on n cores, using the CUDPP library [2], in order to determine the sums of the results in each row. This scan is an efficient method to compute the sums of the multiplications. It takes the array of multiplications as input, plus an array of boolean flags indicating where the results of each row begin. The results of a row together form a *segment* for the scan. Even if many multiplications need to be performed in a number of iterations, such as those in the Jacobi algorithm, the flags array only needs to be computed once at the start, and this can be done in parallel on a GPU. In the scan itself, the sum of the entries in a single segment is computed from right to left, and the intermediate results are written in the array, overwriting the multiplication results. In this way, the final results for the segments are written at the positions of their first entries. The following example illustrates this:

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

Finally, on n cores, the results of the scan are extracted and used for the final computation, which differs between methods (solving system of linear equations, probabilistic bounded until check, etc.). Among the many algorithms available for solving systems of linear equations, we chose to investigate two parallel versions of Jacobi, because Jacobi allows straightforward parallelization, since the computations done within a single Jacobi iteration do not depend on each other. Comparing M_1 and M_2 with each other, M_2 exploits more GPU cores, since the multiplication work is distributed more rigorously (in the first step).

Table I
PARALLEL METHODS IN GPU-PRISM (N.A. = NOT APPLICABLE)

	parallel method	algorithm	M_1	M_2
1	reachability in prob. reward models	Jacobi	√	√
2	probabilistic until checks	Jacobi	√	√
3	stoch. steady state checks	Jacobi	√	√
4	non-det. bounded until checks	bounded mult.		√
5	prob. bounded until checks	bounded mult.		√
6	stoch. bounded until checks	bounded mult.		√
7	set *flags* array	init. for scans	N.A.	√
8	matrix diagonal modification	used with 6		√
9	matrix uniformisation	used with 6		√
10	set vector elements to 0	used with 6		√
11	compute sum of two vectors	used with 6		√

One feature playing a very important role in the achieved speedups is the parallel termination checking for the Jacobi method; at the end of each iteration k, the algorithm checks whether conversion to some small ϵ has taken place, i.e. whether for all i, $|x_i^k - x_i^{k-1}| < \epsilon$, with x_i^k the i-th entry in the (intermediate) result of iteration k. By checking this on n cores at the end of each iteration, copying the intermediate results to the CPU main memory and back, which is the main performance bottleneck for algorithms employing GPUs, can be avoided. Inclusion of parallel termination checking meant a speedup of M_2 of up to 61%. Other aspects have also been parallelized; Table I lists all parallel methods in GPU-PRISM. The top half lists the main methods, while the bottom half lists the supporting methods, consisting of the previously mentioned method to initialise the *flags* array, and some basic vector and matrix manipulations for stochastic bounded until checks.

Using M_1, we also investigated the possibility to use several GPUs in parallel. To utilize g GPUs, the matrix A is split up in a way that n/g rows reside on each GPU. The vector x is copied to all GPUs and space for a partial resulting vector b is reserved on each GPU which computes n/g entries of it. The Jacobi algorithm relies on switching x and b after each iteration which can be assured by switching the pointers to the vectors residing on a single GPU. In a multi GPU environment, all parts of b have to be merged after each iteration by copying them between the GPUs. Although copying the data is a time consuming step, the usage of multiple GPUs enables GPU-PRISM to

Table II
RESULTS FOR VERIFYING PROPERTY 1 OF THE HERMAN, CLUSTER, AND TANDEM PROTOCOL

protocol	seq. time	par. time M_1		par. time M_2
		1 GPU	2 GPUs	1 GPU
herman 15	10.54s	12.84s	12.14s	3.99s
cluster 464	4,270.26s	643.85s	1,024.34s	807.42s
tandem 2,047	3,642.62s	384.68s	946.76s	279.65s

Table III
RESULTS FOR VERIFYING PROPERTY 3 OF THE TANDEM PROTOCOL

protocol	seq. time	par. time M_2
tandem 255	0.60s	0.79s
tandem 511	7.42s	4.48s
tandem 1,023	34.14s	8.20s
tandem 2,047	268.31s	54.31s

check larger problems and significant speedups can still be achieved. This approach can also be utilized to check large problems on a single GPU by copying the partitioned A sequentially to this GPU. Here the copying of data slows down the computation to a level where the sequential CPU method is faster.

III. EXPERIMENTS

We evaluated GPU-PRISM on several case studies from the standard distribution of PRISM (cf. [1]). The experiments were performed on (one core of) a personal computer with Intel Core i7 CPU 920 running at 2.67GHz with 8 CPU cores and 12 GB RAM. We used two NVIDIA geForce 285 GTX (MSI) graphics cards with 1 GB VRAM and 240 streaming processors each running at 0.6 GHz. The computer was running Ubuntu 9.04 with CUDA 2.2 SDK and the NVIDIA driver version 195.36.24. Table II presents the runtime results for verifying property 1 of instances of the herman, cluster, and tandem protocols. (All these models and their corresponding property files are part of the standard distribution of PRISM.) This property requires solving a system of linear equations. The advance of using the GPU for computing the matrix vector multiplication can be clearly seen for both methods. Comparing the usage of one or two GPUs in M_1 reveals the slowdown imposed by copying parts of b between the devices, still a speedup of nearly 4 is achieved. While M_2 is faster for herman and tandem it can not cope with M_1 in the cluster protocol. This can be explained by the density of the matrix A, the cluster protocol consists of a sparser matrix where more of the $n \times n$ threads are idle while the first stage.

Table III contains the runtimes for verifying a stochastic bounded until property (property 3 in the corresponding property file) of instances of the tandem protocol. Here we see a direct correlation of the complexity of the model and the speedup achieved by using a GPU. This proves our assumption of copying being the bottleneck. The speedup achieved by parallel computation can not recompense the time needed for copying.

IV. CONCLUSION AND FUTURE WORK

GPU-PRISM exhibits much faster runtimes when matrix-vector multiplications are involved. In the future the tool will develop towards using better algorithms for the crucial linear algebraic operations. There are at least two directions in this context: 1) a more efficient use of a single GPU by employing as much as possible processors in parallel, and 2) use of multiple GPUs. We tested GPU-PRISM on two GPUs with significant improvements that we report in the extended version of [1]. The algorithm that we use is easily scalable for multiple GPUs.

GPU-PRISM is available from the authors on request.[1]

REFERENCES

[1] D. Bošnački, S. Edelkamp, and D. Sulewski. Efficient Probabilistic Model Checking on General Purpose Graphics Processors. In Proc. 16th International SPIN Workshop, LNCS 3925, pp. 32–49, Springer, 2009. Extended version submitted to STTT.

[2] CUDA Data Parallel Primitives Library. http://gpgpu.org/developer/cudpp

[3] CUDA Programming Forum. http://www.nvidia.com/object/cuda_home.html

[4] 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.

[5] 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.

[1] Once licence issues related to standard PRISM are resolved, we will make GPU-PRISM available via a tool web page.