A GPGPU–based Simulator for `Prism`: Statistical Verification of Results of PMC [extended abstract]

Marcin Copik[1] and Artur Rataj[2] and Bożena Woźna-Szcześniak[3]

[1] RWTH Aachen University,
Schinkelstr. 2, 52062 Aachen, Germany
mcopik@gmail.com
[2] IITiS, Polish Academy of Sciences
ul. Bałtycka 5, 44-100 Gliwice, Poland
arturrataj@gmail.com
[3] IMCS, Jan Długosz University
Al. Armii Krajowej 13/15, 42-200 Częstochowa, Poland.
b.wozna@ajd.czest.pl

Abstract. We describe a GPGPU–based Monte Carlo simulator integrated with `Prism`. It supports Markov chains with discrete or continuous time and a subset of properties expressible in PCTL, CSL and their variants extended with rewards. The simulator allows an automated statistical verification of results obtained using `Prism`'s formal methods.
Keywords: GPGPU, Monte Carlo simulation, `Prism`, probabilistic model checking, statistical model checking, probabilistic logics.

1 Introduction

We present a GPGPU–based simulator which extends the model checker `Prism` [9]. The simulator uses the Monte Carlo method for a statistical probabilistic model checking [14, 10] (SPMC). SPMC involves a generation of a large number of random paths (i.e. samples) in a probabilistic Markov chain, evaluating a given property on each path, and finally finding an average of these evaluations, which approximate a correct value of the property. Monte Carlo methods typically are able to precisely compute confidence intervals (CI) around the approximated value.

The GPGPU simulator (further called $\mathcal{S}^G$) is integrated with `Prism`, which allows to check a single model implementation using either one of `Prism`'s probabilistic model checking (PMC) methods, or $\mathcal{S}^G$. $\mathcal{S}^G$ supports the same models and properties as the `Prism`'s CPU–based simulator (further denoted $\mathcal{S}^C$), yet the latter, lacking GPGPU acceleration and on–the–fly compilation of the model, is considerably slower.

Beside the simulator itself, we present its simple application: an automated method of verifying property values computed using `Prism`'s formal methods. Namely, the user may request, that certain property classes be computed in two steps:

1. A PMC step. A property is computed using one of `Prism` 's formal PMC methods;
2. An automated statistical verification (ASV) step. The obtained property value v is statistically evaluated using $\mathcal{S}^G$; it is then checked if v fits into a number of confidence intervals (CI) of various confidence levels.

Because a significant imprecision in a PMC method is typically caused by some mechanism for reducing computational complexity, the user might tend to disable such a mechanism, rather than wait for a statistical verification. This is why it is crucial that the verifying simulator be fast: it should effectively save user's time, by providing data in tight CI within a short time.

The paper is constructed as follows. Section 2 describes what is currently supported by $\mathcal{S}^G$. In Sec. 3 we provide a description of some implementation details of $\mathcal{S}^G$. In Sec. 4 ASV is discussed Section 5 presents a case study. Finally, the last section concludes the paper.

2 Supported models and properties

$\mathcal{S}^G$ accepts a specific set of properties, for two finite probabilistic classes of Markov chains: a *discrete-time Markov chain* (DTMC) and a *continuous-time Markov chain* (CTMC). Unlike DTMC, where each transition corresponds to a discrete time-step, in a CTMC transitions occur in continuous time given by a negative exponential distribution. Both of the classes can be enriched with rewards structures, resulting respectively in rDTMC and rCTMC. A reward structure allows to specify two distinct types of rewards: state (instantaneous) and transition (cumulative) ones, assigned respectively to states and transitions by means of a reward function. Formal definitions of all of the above systems can be found e.g. in [8].

The temporal logics *Probabilistic Computation Tree Logic* (PCTL) [6] and *Continuous Stochastic Logic* (CSL) [1] can be used to specify properties for respectively DTMCs and CTMCs. $\mathcal{S}^G$ recognises only flat subsets of each logic. We will refer to these subsets as respectively FlatPCTL and FlatCSL.

Definition 1 (Syntax of FlatPCTL). *Let $a \in \mathcal{AP}$ be an atomic proposition, $\sim \in \{<, \leq, \geq, >\}$, $p \in [0,1]$ a probability bound, and k is a non–negative integer or ∞. The syntax of FlatPCTL is defined inductively as follows:*

$$\phi ::= \mathtt{P}_{\sim p}[\psi], \quad \psi ::= \mathtt{X}\phi_1 \mid \mathtt{G}^{\leq k}\phi_1 \mid \mathtt{F}^{\leq k}\phi_1 \mid \phi_1 \mathtt{U}^{\leq k}\phi_1 \mid \phi_1 \mathtt{R}^{\leq k}\phi_1,$$
$$\phi_1 ::= a \mid \phi_1 \wedge \phi_1 \mid \neg\phi_1.$$

In the syntax above, we distinguish between state formulae ϕ, ϕ_1 and path formulae ψ, which are evaluated over states and paths, respectively. A property of a model is always expressed as a state formula. The path modalities (i.e., *next state* – $\mathtt{X}$, *bounded globally* – $\mathtt{G}$, *bounded eventually* – $\mathtt{F}^{\leq k}$, *bounded until* – $\mathtt{U}^{\leq k}$, and *bounded release* – $\mathtt{R}^{\leq k}$), which are standard in temporal logics, can occur only within the scope of the *probabilistic operator* $\mathtt{P}_{\sim p}[\cdot]$.

Intuitively, a state s satisfies $\mathtt{P}_{\sim p}[\psi]$ if the probability of taking a path from s satisfying path formula ψ meets the bound $\sim p$. Next, $\mathtt{X}\phi$ is true if ϕ is satisfied in the next state; $\mathtt{G}^{\leq k}\phi$ is true if ϕ holds for all time-steps that are less or equal to k; $\mathtt{F}^{\leq k}\phi$ is true if ϕ is satisfied within k time-steps; $\phi_1 \mathtt{U}^{\leq k}\phi_2$ is true if ϕ_2 is satisfied within k time-steps and ϕ_1 is true from now on until ϕ_2 becomes true. $\phi_1 \mathtt{R}^{\leq k}\phi_2$ is true if either ϕ_1 is satisfied within k time-steps and ϕ_2 is true from now on up to the point where ϕ_1 becomes true, or

ϕ_2 holds for all time-steps that are less or equal to k. The formal semantics over DTMC can be found e.g. in [6, 8].

$\mathcal{S}^G$ supports also an extension of FlatPCTL allowing specifications over reward structures by means of the following state formulae: $\mathtt{R}_{\sim r}[\mathtt{C}^{\leq k}] \mid \mathtt{R}_{\sim r}[\mathtt{I}^{=k}] \mid \mathtt{R}_{\sim r}[\mathtt{F}\phi]$ where $\sim \in \{<, \leq, \geq, >\}$, $r \in \mathbb{R}_{\geq 0}$, $k \in \mathbb{N}$, and ϕ is a FlatPCTL formula.

The formal semantics over rDTMC can be found in [8]. Here we only provide an intuition. Namely, a state s of an rDTMC satisfies $\mathtt{R}_{\sim r}[\mathtt{C}^{\leq k}]$, if from state s the expected reward *cumulated* after k time-steps satisfies $\sim r$. Next, a state s of an rDTMC satisfies $\mathtt{R}_{\sim r}[\mathtt{I}^{=k}]$, if from state s the expected state reward at time-step k satisfies $\sim r$. Finally, a state s of an rDTMC satisfies $\mathtt{R}_{\sim r}[\mathtt{F}\phi]$, if from state s the expected reward cumulated before a state satisfying ϕ is reached meets the bound $\sim r$.

Definition 2 (Syntax of FlatCSL). *Let a and p be as in Definition 1, and I be an interval of $\mathbb{R}_{\geq 0}$. The syntax of FlatCSL is defined inductively as follows:*

$$\phi ::= \mathtt{P}_{\sim p}[\psi], \quad \psi ::= \mathtt{X}\phi_1 \mid \mathtt{G}^I\phi_1 \mid \mathtt{F}^I\phi_1 \mid \phi_1\mathtt{U}^I\phi_1 \mid \phi_1\mathtt{R}^I\phi_1, \quad \phi_1 ::= a \mid \phi_1 \wedge \phi_1 \mid \neg\phi_1.$$

Satisfying $\mathtt{P}_{\sim p}[\psi]$ and path modalities are the same for FlatCSL as for FlatPCTL, except that the parameter of the modalities is an interval I of the non-negative reals, rather than an integer upper bound. For example, the path formula $\phi_1\mathtt{U}^I\phi_2$ holds if ϕ_2 is satisfied at some time instant in the interval I and always earlier ϕ_1 holds.

$\mathcal{S}^G$ supports an extension of FlatCSL allowing specifications over reward structures in a manner similar to FlatPCTL, the only difference are the time bounds: $\mathtt{R}_{\sim r}[\mathtt{C}^{\leq t}] \mid \mathtt{R}_{\sim r}[\mathtt{I}^{=t}] \mid \mathtt{R}_{\sim r}[\mathtt{F}\phi]$ where $\sim \in \{<, \leq, \geq, >\}$, $r,t \in \mathbb{R}_{\geq 0}$, and ϕ is a FlatCSL formula.

The extension of CTMC with rewards is analogous to that of DTMC, given above, barring the mentioned differences in time bounds. The formal semantics over rCTMC can be found in [8], see though that $\mathcal{S}^G$does not support therein mentioned steady state.

3 Implementation of $\mathcal{S}^G$

In a case of Markov models implemented in the `Prism` language, a generation of random simulation paths is not computationally expensive. A single transition consists of an evaluation of its guards, an enumeration of updates if viable, a random selection of subsequent transitions and finally an estimation of properties. The syntax of guards and updates allows simple arithmetical and logical operations and a few basic mathematical functions, such as a power or a logarithm. `Prism` comes already with the mentioned CPU–based simulator $\mathcal{S}^C$, well–suited to debugging tasks but slow, as it is sequential and generates each path by reinterpreting a model specification.

Instead of such an on–the–fly reinterpretation, the tool Ymer [15] compiles expressions to a form which is faster to evaluate repeatedly. Another tool APMC [7], in, turn provides a translation of `Prism` models to C programs which are later compiled and executed.

Another obstacle preventing the $\mathcal{S}^C$ from achieving a reasonable performance is its inherent sequentiality. A Monte Carlo simulation is considered to be embarrassingly parallel – it samples the model by generating a large number of independent random

paths. Prism has approached this problem by providing the ability to perform distributed sampling, Ymer and APMC support distributed sampling as well. The latest version of Ymer implement multi–threaded sampling as well [16].

The improvement in parallel and distributed sampling is limited by the number of threads supported on multi–core CPU systems. A processor with a rich set of instructions and multiple cache levels is a perfect tool for complex and general problems but using it for a simple simulation of a moderate Markov model would be a very expensive over–engineering, both financial cost– and energy–wise. On the contrary, recent advances in GPGPUs made them a very efficient replacement for such computations, which benefit from massive parallelism. This simultaneous execution of hundreds and thousands lightweight threads on a GPGPU comes at a price: well-known limitations include a restriction of a group of threads to execute the same instruction at the same time or a burdensome memory model with a high cost of non–regular memory access patterns. We believe though, that those restrictions do not play a significant role in a Monte Carlo simulation of Prism models. For that purpose we have chosen OpenCL [13] as a framework and programming language for GPGPU simulation.

3.1 Architecture

Our decision to implement the simulator engine for Prism using OpenCL has been driven by its capability of supporting many types of devices offered by different vendors, the most common being graphic cards and multi–core CPU servers. We will further use an OpenCL–specific terminology.

To simplify the implementation we have used OpenCL bindings for Java, the main source language of Prism, provided through the JavaCL library [2].

Fig. 1 presents a general scheme of the simulator. Within Prism there is no significant difference between starting a simulation in using $\mathcal{S}^C$ or $\mathcal{S}^G$. In both cases a model and a list of properties is required. Then, a just–in–time source-to-source translation of the model and properties produces a dedicated *compute kernel* (block Kernel generator in Fig. 1) which is basically a program for a number of vector processors, which use a single–instruction–multiple–data paradigm. The approach is very different from the reinterpretation scheme implemented in $\mathcal{S}^C$, which stores the model and properties in memory structures, and not as an automatically generated program. $\mathcal{S}^G$ executes kernels implemented in OpenCL C language, a modified version of C99, which has been adapted to OpenCL's device abstraction and stripped from features usually not allowed on a device, such as recursion or function pointers. Those simple programs can be effectively compiled into native bytecode of the device (typically, a graphic card). The syntax of Prism language makes the translation process rather straightforward due to the similarity between its expressions and the OpenCL C language. Only minor and automatically applied changes allow to obtain an expression valid in the latter language.

3.2 Method

A scheme of generating a single random path (sample) is presented in Algorithm 1. Capitalised identifiers indicate constants which are injected into kernel source. Many

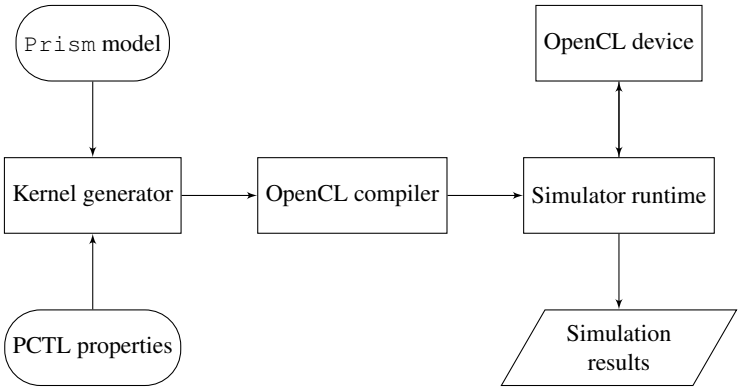

Fig. 1. The OpenCL-based simulator engine in `Prism`.

details have been omitted in the case of continuous time models. For example, in CTMC both times of entering and leaving state may be required in certain situations, such as a bounded until or a property with cumulative reward. The fargument `idx` is an unique identifier of kernel instance, `offset` specifies how many paths have been processed in kernels previously computed on the device. The argument `seed` seeds the generator of pseudorandom numbers. As the scheme is an equivalent of an OpenCL kernel, the two last arguments are OpenCL–specific storage buffers in the device memory, where the kernel is allowed to save results of property verification, and also the length of created path used to display sampling statistics. The first loop (lines 5–7) resets the collected statistical data about properties, the second loop (lines 8–27) generates the path until any of the following: its maximum length is reached (line 8), a deadlock (line 14) or a self–loop (line 18) occurs, or all properties are verified precisely enough (line 24). The last loop (lines 29–31) copies the collected data into the global memory.

The implementation had to be specially adapted for GPGPU devices, given that there can be a significant slow–down if the kernel diverges from the SIMD paradigm. Another example of such change is choosing always the smallest, most space efficient integer type for holding a state variable, which is possible as `Prism` models specify ranges for each such variable. For efficiency, if a simulated model updates a variable with a value exceeding these ranges, then the behaviour is undefined. A large speed–up can be achieved by handling an update synchronised between `Prism` modules in one step. If such update is performed on the same copy of state vector, it may induce a race condition of the type Read-After-Write. $\mathcal{S}^G$ detects such situations and creates additional copies of the affected variables, if necessary, instead of relying on the OpenCL compiler, which might handle the issue less effectively. Creating only one instance of a state vector is crucial for performance because it decreases significantly memory space used by the kernel, as discussed later with memory complexity of the algorithm. The simulation kernel is also capable of detecting when there is only one transition available, and it does not change the state. Such a behaviour indicates that there is only a single self–loop in the current state, which is interpreted by $\mathcal{S}^G$ as a stop condition. If there is an unbounded property which has not been satisfied yet, its value is not going

to change. If there is a property with a lower bound, which has not been reached yet, it can be evaluated immediately and the process of simulating the current path ends.

Our pseudorandom number generator of choice is Random123 [12], which provides a performance satisfying our needs. For more details on model conversion into a form for GPGPU computation see [3].

Algorithm 1 A generic path generation algorithm for OpenCL kernel.

```
1:  procedure KERNEL(idx, offset, seed, path_lengths, property_results)
2:      prng ← initialize_prng(seed, idx, offset)          ▷ A distinct seed for each path
3:      state_vector ← INITIAL_STATE_VECTOR
4:      time ← 0
5:      for each property p ∈ PROPERTIES do
6:          reset(results_p)
7:      end for
8:      for i < MAX_PATH_LENGTH do
9:          active_updates ← evaluate_guards(state_vector)      ▷ Single and synchronised
10:         time_update(time)                                   ▷ More complex for CTMC
11:         for each property p ∈ PROPERTIES do
12:             active_properties ←property_p_update(state_vector, results_p, time)
13:         end for
14:         if active_updates = 0 then
15:             break                                           ▷ Deadlock detected
16:         end if
17:         no_change ← update(prng, state_vector, active_updates)
18:         if no_change ∧ active_updates = 1 then
19:             for each property p ∈ PROPERTIES do
20:                 active_properties ←property_p_update(state_vector, results_p, time)
21:             end for
22:             break                                           ▷ Loop detected
23:         end if
24:         if ¬active_properties then
25:             break                                           ▷ Stop sampling
26:         end if
27:     end for
28:     path_lengths[idx + offset] = i                  ▷ Save results in global memory
29:     for each property ∈ property_results do
30:         property[idx + offset] = results_p          ▷ Save results in global memory
31:     end for
32: end procedure
```

A generated kernel is passed as a string of source code to a specific device compiler (block `OpenCL compiler` in Fig. 1). This compilation does not add a significant overhead on modern OpenCL platforms – we have found that it typically takes less than one second for tested models. A kernel represented in device bytecode is sent to simulator's runtime (see again Fig. 1), where a range of *work items* is enqueued on the device, as described in more details in the next section. Each one of them is responsible

for producing exactly one random path through the model and its identifier *idx*, is paired with an *offset*, in order to produce a unique key across many separate kernel enqueued on the device. The key is necessary for correct accessing memory storage and unique random seeds for each generated path.

4 Automatic statistical verification

The ASV step is optionally triggered at the user request, in one of the following ways:

- unconditionally;
- if a quantitative property is being computed, like the probability value;
- if a steady state is detected prematurely in an iterative PMC method.

The last criterion is discussed in more detail in the example in Sec. 5.

The ability to process an extensive number of samples in a very short time often allows for a reliable and fast ASV of a property value v obtained using PMC. In the ASV step, in order to save the user's time, $\mathcal{S}^G$ must finish within a predefined time $T_{\max}$.

After the ASV step, the user is presented with a set of diagnostics, so that he can estimate the correctness of v. Let the simulator estimate a value w of the same property. Let R_{CI} be the ratio of the width of CI at confidence level 90% to w. The following independent diagnostics can be presented:

- $T_{\max}$ too low to reach $R_{CI} < R^{CI}_{\max}$; $R^{CI}_{\max}$ has a default value of $1 \cdot 10^{-2}$ and can be customised;
- v within a CI of a confidence level x, outside a CI of a confidence level y, where $x \in L = \{90\%, 95\%, 99\%\}$, $y \in L \vee \{< 90\%\}$.

5 Case study

An instantaneous reward in a CTMC can be estimated by weighting the reward function over a probabilistic distribution of states at a time instant t. `Prism` computes the distribution by uniformisation (Jensen's method) [5] which discretises the CTMC with respect to a constant rate. Then, probabilities are approximated by a finite summation Z of Poisson–distributed steps of the derived DTMC. The number of these steps depends on the precision required, and is computed using the Fox–Glynn method [4]. Yet, in order to shorten computation time, when performing that summation, `Prism` also tests, if a steady state has been reached, by finding a maximum difference, either relative or absolute, between elements in solution vectors from two successive steps. If the difference is smaller than a constant threshold ε, the summation is terminated early.

`Prism`'s default criterion of the termination is to use a relative difference and $\varepsilon = 10^{-6}$. The criterion can be customised in order to set a compromise between precision and computational complexity.

To illustrate the ASV, we will discuss a model where the detection of a steady state is premature if the default termination criterion is used. In effect, were the automatic SV not enabled, the user would obtain incorrect results without a warning.

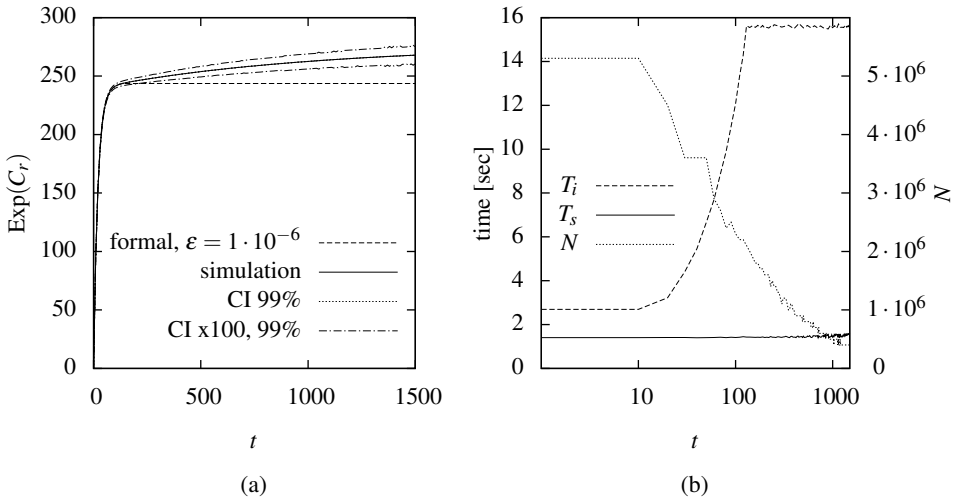

Fig. 2. (a) Estimates of $\mathrm{Exp}(C_r)$, CIs are narrow enough to be hardly seen, thus their magnification by 100 is also show. (b) elapsed CPU time and the number of samples per single property, T_i and T_s are total times of, respectively, the formal method performing Z, and the statistical estimation on $\mathcal{S}^G$, N is the number of samples chosen by $\mathcal{S}^G$ in order to fit within $T_{\max}$.

We will use a modified Model 2 from [11]. It is a simple CTMC with multiple clients and servers, in which some of the servers can occasionally be broken. To trigger the said premature termination of Z, we modify the model by using constants $r_s = 1, r_l = 500$ (see [11] for details), i.e. the server is slower at initiating a connection with a client, but faster at processing a request. We ask for an expected instantaneous reward value C_r, equal to the number of clients requesting at a given time instant t.

Let the user choose the default termination criterion, and let he also request, that SV be used for up to $T_{\max} = 1.5$ sec. after a PMC step such that a steady state detection has terminated early Z (or any other formal iterative method which uses ε). $\mathrm{Exp}(C_r)$ against t, found using both a formal PMC and $\mathcal{S}^G$, is depicted in Fig. 2(a). We see that the model undergoes a fast change at $t \approx 100$, during which the increase of requests becomes rapidly slower. The constant $T_{\max}$ allows for a fairly narrow CI at 99% confidence level – its width never reaches 0.1. In the case of the discussed diagram $R_{CI} < 4 \cdot 10^{-4}$ for any t – such a narrow CI makes it probable that following the said change at $t \approx 100$, the PMC results become less and less precise. This would trig respective warnings, that values from the results of the PMC step fall outside a CI of a high confidence level.

The relative temporal overhead of SV for the chosen $T_{\max}$ is illustrated in Fig. 2(b). It is largest for small t and makes `Prism` run for about 50% longer. For $t \gtrsim 200$ `Prism` needs less than 10% of an additional CPU time to perform the SV step. The figure also shows the number of samples which can be computed within $T_{\max}$; wee see that the number decreases for larger t due to longer paths which need to be generated by the simulator.

In order to obtain the CPU times in Fig. 2(b), we have used an AMD R9 Nano, a modern mid-range GPU with 4096 stream processors. Let us compare that GPU to

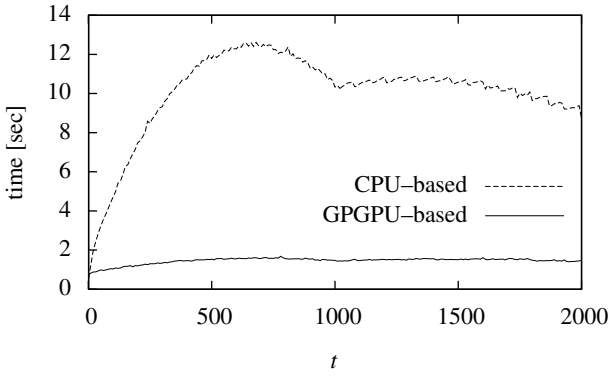

Fig. 3. Computation time on two OpenCL devices.

another OpenCL device, containing two Intel Xeon E5-2630 v3 CPUs with clock frequency 2.40GHz, each of them providing 8 cores with HyperThreading, resulting in 32 threads for OpenCL. Fig. 3 compares the simulation time from Fig. 2(b) to that of the CPU OpenCL device, both evaluating the same number of samples. In the case of the latter device, we see times in the range of 8 and 12 seconds, i.e. it is several times slower. This shows the advantage of streams processors in the case of a large number of independent, non–memory intensive tasks.

$\mathcal{S}^C$ has been excluded from the chart as it is not optimised for speed. At $t = 2000$, where long paths make it especially advantageous to use the on–the–fly compilation, it took $\mathcal{S}^C$ over 130 minutes to evaluate the same reward property on the same CPU, thousands of times slower comparing a respective simulation on a GPGPU.

6 Discussion

The SV limits itself now to diagnostics, but it could be straightforwardly extended to influence on the PMC step. For example, if a property p_i turns out to be computed imprecisely in the PMC step, and the following property to compute p_{i+1} differs only in the time instant t, the SV could automatically decrease ε for the computation of p_{i+1}.

We expect to release an open-source version of the tool in the following months. The further development would be focused on a parallelisation across multiple OpenCL devices, with a dynamic and automatic load balancing.

References

1. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model-checking continuous-time markov chains. Transactions on Computational Logic (TOCL) **1**(1), 162–170 (2000)
2. Chafik, O.: Javacl. `https://github.com/nativelibs4java/JavaCL` (2016)
3. Copik, M.: GPU-accelerated stochastic simulator engine for PRISM model checker. Bachelor's Thesis, Silesian University of Technology, Gliwice, Poland (2014)

4. Fox, B.L., Glynn, P.W.: Computing poisson probabilities. Commun. ACM **31**(4), 440–445 (1988)
5. Gross, D., Miller, D.: The randomization technique as a modeling tool and solution procedure for transient Markov processes. Operations Research **32**(2), 926–944 (1984)
6. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal aspects of computing **6**(5), 512–535 (1994)
7. Herault, T., Lassaigne, R., Peyronnet, S.: Apmc 3.0: Approximate verification of discrete and continuous time markov chains. In: Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems, QEST '06, pp. 129–130. IEEE Computer Society, Washington, DC, USA (2006). DOI 10.1109/QEST.2006.5. URL http://dx.doi.org/10.1109/QEST.2006.5
8. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07), *LNCS*, vol. 4486, pp. 220–270. Springer (2007)
9. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: G. Gopalakrishnan, S. Qadeer (eds.) Proc. 23rd International Conference on Computer Aided Verification (CAV'11), *LNCS*, vol. 6806, pp. 585–591. Springer (2011)
10. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: International Conference on Runtime Verification, pp. 122–135. Springer (2010)
11. Pourranjbar, A., Hillston, J., Bortolussi, L.: Don't just go with the flow: Cautionary tales of fluid flow approximation. In: Computer Performance Engineering - 9th European Workshop, pp. 156–171 (2012)
12. Salmon, J.K., Moraes, M.A., Dror, R.O., Shaw, D.E.: Parallel random numbers: As easy as 1, 2, 3. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, pp. 16:1–16:12. ACM, New York, NY, USA (2011). DOI 10.1145/2063384.2063405. URL http://doi.acm.org/10.1145/2063384.2063405
13. Stone, J.E., Gohara, D., Shi, G.: Opencl: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering **12**(1-3), 66–73 (2010)
14. Younes, H., Kwiatkowska, M., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking. International Journal on Software Tools for Technology Transfer (STTT) **8**(3), 216–228 (2006)
15. Younes, H.L.S.: Ymer: A statistical model checker. In: Proceedings of the 17th International Conference on Computer Aided Verification, CAV'05, pp. 429–433. Springer-Verlag, Berlin, Heidelberg (2005). DOI 10.1007/11513988_43. URL http://dx.doi.org/10.1007/11513988_43
16. Younes, H.L.S.: Ymer. https://github.com/hlsyounes/ymer (2016)