Speeding Up Parametric Model Checking Using GPGPU Computation

Marcel Erkko
University of Twente
PO Box 217, 7500 AE Enschede
the Netherlands

m.a.erkko@student.utwente.nl

ABSTRACT

GPGPU-based parallelisation has gained popularity due to its capability of efficiently handling compute-intensive tasks. The prerequisite is that the tasks being performed must be parallelisable using clever data structures and proper synchronisation. Parallelisation on graphics processing units (GPUs) has found applications in machine learning, computer vision, and statistical analysis. Within statistical analysis, one such compute-intensive task is the model checking of parametric Markov chains. This research proposes a parallelised approach to the evaluation phase of parametric model checking of Markov chains.

Keywords

Markov chain, parametric model checking, GPGPU, CUDA, interval arithmetic

1. INTRODUCTION

When dealing with statistical analysis, GPUs have been found to be much faster than single thread applications. The multitude of cores on GPUs allow for simultaneous computation of hundreds of inputs, producing speedups ranging from 100 to 1000 times [15]. Parametric models are a family of probability distributions with a finite number of parameters such that each instantiation of the parameters produces a non-parametric stochastic model [6]. When utilising parametric models with interval values, the result can be guaranteed to be within a specific bound based on the instantiation of the parameters. Parametric model checking (PMC) is a highly parallelisable task, as the model is representable as a large rational function [5]. The entire parameter space – or combination of parameter ranges – can be evaluated concurrently using general-purpose GPU (GPGPU) parallelisation and one can quickly find every combination of parameter instantiations which fulfils a given property or result bound.

The analysis and optimisation of parametric Markov models has received much attention during recent years [5]. However, the evaluation of the parameter space for complex parametric models is still a slow process. CUDA (Compute Unified Device Architecture) is a GPGPU-based parallelisation framework produced by NVIDIA. This framework is specifically designed for Nvidia GPUs to perform compute intensive portions of programs on thousands of GPU cores in parallel. CUDA's organisation of threads fits well with interval arithmetic, as the kernel is organised into blocks of threads. During implementation of the parallel evaluator, it was found that utilising CUDA's thread indexing method is a highly efficient way to evaluate interval parameters. However, simply using the inbuilt indexing method of CUDA limits the number of parameters to one to three parameters.

This research aims to improve the speed of the evaluation phase of PMC using this parallelisation framework by evaluating the entire parameter space in a maximally concurrent manner. The following research questions will be considered in this paper.

- How much of a speed-up is provided by using GPGPU parallelisation in model checking of (discrete-time) parametric Interval Markov Chains?
- What are the optimal kernel dimensions for model checking parametric Markov Chains?

The research performed is an extension of the work done by Gainer et al. [2018] and their optimised construction of parametric Markov Chains. The goal is to implement a CUDA version for the evaluation phase of parametric model checking based on their programmatic tool for model checking. The results of the parallelised program are then compared against a single-threaded version of the same instructions representing the parametric model. What this research does *not* intend to do is optimise or change the parametric model produced in any way, but rather to parallelise the evaluation of it in an optimal manner.

Following sections will provide background regarding related work and preliminary information about parametric models, CUDA, and interval arithmetic. Following that will be a discussion about the transformation of the single-threaded program into a parallelised version and how the CUDA program was optimised. Lastly, the parallel version is compared against the single-threaded approach in model checking of the entire parameter space and the results and conclusions are addressed.

2. RELATED WORK

As mentioned previously, this research aims to continue the work done in the paper by Gainer et al. [2018]. In their paper, the parametric model is represented as a finite automaton, which is then utilised to construct a representation of a rational function. The output is a Directed Acyclic Graph (DAG) structure for function evaluation, with the function being represented in the form of an arithmetic circuit. This approach already provided a significant speed-up in comparison to other modern parametric model checkers [5].The ePMC tool created by Gainer et al. will be extended by transforming the generated list of instructions into equivalent CUDA instructions and comparing against their interval implementation written in C using a single-threaded approach for evaluation.

Parametric Interval Markov Chains (pIMCs) are a type of parametric model which provide various benefits. Research into pIMCs has been done by Bart et al. [2017], showing that the combination of parametric Markov Chains (pMCs) and interval Markov Chains (IMCs) is strictly more expressive than the other two individually. That is, anything that can be expressed

36th Twente Student Conference on IT, Febr. 4th, 2022, Enschede, The Netherlands.

by a pMC and IMC, can be expressed by pIMCs, while the opposite does not hold [2]. Their work provides validation for the expressiveness and usefulness of pIMCs.

CUDA programming has found use in the implementation of Markov Chain Monte Carlo simulations and statistical analysis using mixture modelling approaches [15]. Although the results of the former showcase speedups from 100 to 1000 times faster on certain highly structured Bayesian models, Suchard et al. [2010] note that the time investment necessary in developing new programming skills has been a barrier in the adoption of GPGPU parallelisation techniques within the statistics community. During the literature review it was found that there is a scarcity of research into parallelisation of parametric model checking. Therefore, this paper aims to provide further proof of the usefulness of parallelisation in statistical analysis.

3. PRELIMINARIES

3.1 Non-Parametric & Parametric Models

Non-parametric models are a type of statistical model where each transition has a real valued probability attached. Parametric models are a generalised version of this, where the transitions are modelled as a function that are dependent on the parameters. Knuth's dice [10] is a famous example of Markov Chains where the model emulates the throw of a 6-sided dice using a coin. **Figure 1** displays the throwing of the dice emulated using a fair (left) or biased coin (right), or a non-parametric and parametric model respectively.

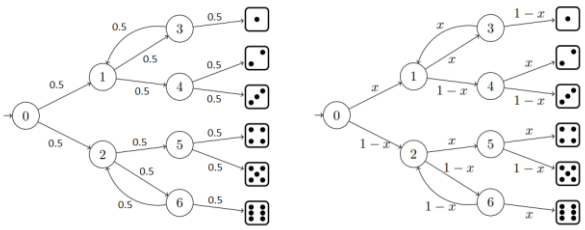

Figure 1. Non-parametric and parametric Knuth's Dice. [5]

Markov Chains (MC) are stochastic models characterised by the property that the future states of the process are independent of the past states. They have been applied in for example communications, automatic control, signal processing, and economics [3]. Although this can be modelled as a non-parametric model, Junges et al. [2021] state that utilising a parametric Markov Chain allows for a realistic representation of the system state by coupling transitions, therefore inducing global restrictions on the possible probability distributions. Parametric Interval Markov Chains are an extension of this, where the parameters are represented as interval values, providing benefits to expressiveness and accuracy of the models representativeness. Evaluation of the model also produces an interval, with a guarantee that the result is within this interval's bounds. This can then be utilised to compare against lower and upper bounds restrictions that one is interested in. The starting point for this research was a list of interval arithmetic instructions written in C, which represents the rational functional that describes the parametric model.

3.2 CUDA Programs

CUDA programs are generally written in C or C++, with the functions executed on the GPU being called kernels. Threads in CUDA are organised in blocks, namely recognised as thread blocks, which all reside on the same processor core and share the same memory resources. The blocks are further organised into one-dimensional, two-dimensional, or three-dimensional grids of thread blocks. The blockIdx - a built-in variable - is used to identify each block within the grid, while the dimension of the thread block is accessible through the blockDim variable. Both values are accessible within the kernel and these thread blocks are independent, meaning one can execute them in any order - in parallel or in series - thus allowing for code that scales with the number of cores of the GPU [11].

Features supported by CUDA are dependent on the compute capability of the hardware the program is compiled and ran on. Additionally, NVIDIA GPUs are organised as an array of Streaming Multiprocessors (SMs). These multiprocessors create, manage, schedule, and execute threads in groups of 32 parallel threads called warps. The maximum number of warps per SM – or the maximum instruction throughput - is dependent on the compute capability of the GPU and limited by various hardware related maximum properties such as the number of blocks per SM, warps per block, and warps per SM [11].

When CUDA programs invoke a kernel from host code – or CPU code - the thread blocks are distributed to multiprocessors with available execution capacity. Thus, the parallelisability of a CUDA program is additionally dependent on this hardware limitation, as it dictates factors such as maximum thread block size, maximum shared memory per block, and maximum registers utilisable per thread and thread block [11].

3.3 Interval Arithmetic

Interval arithmetic is a mathematical technique used to solve uncertainty problems which cannot be efficiently solved using floating-point arithmetic. As floating point numbers contain a limited number of bits, scientific computations can contain minor accumulative errors due to certain values not being possible to be represented using floating-point numbers [4]. In interval arithmetic, instead of representing a value as a single number, each value is represented as a range of possibilities containing a real value x in it. This allows for additional flexibility and consideration for errors during computation of long lists of arithmetic instructions.

The rational functions produced by the ePMC tool are constructed using the arithmetic operations of addition, additive inverse, multiplication, and multiplicative inverse [5]. These operations can all be performed using interval arithmetic, but with slight modifications. The operations used in parametric models are defined below in **Figure 2**.

$$[x_1, x_2] + [y_1, y_2] = [x_1 + y_1, x_2 + y_2] \quad (1)$$

$$[x_1, x_2] - [y_1, y_2] = [x_1 - y_2, x_2 - y_1] \quad (2)$$

$$[x_1, x_2] \cdot [y_1, y_2] = [\min\{x_1y_1, x_1y_2, x_2y_1, x_2y_2\}, \max\{x_1y_1, x_1y_2, x_2y_1, x_2y_2\}] \quad (3)$$

$$\frac{1}{[y_1, y_2]} = \left[\frac{1}{y_2}, \frac{1}{y_1}\right] \quad \text{if } 0 \notin [y_1, y_2] \quad (4)$$

$$\frac{1}{[y_1, 0]} = \left[-\infty, \frac{1}{y_1}\right] \quad (5)$$

$$\frac{1}{[0, y_2]} = \left[\frac{1}{y_2}, \infty\right] \quad (6)$$

$$\frac{1}{[y_1, y_2]} = \left[-\infty, \frac{1}{y_1}\right] \cup \left[\frac{1}{y_2}, \infty\right] \subseteq [-\infty, \infty] \quad \text{if } 0 \in (y_1, y_2) \quad (7)$$

Figure 2. Interval arithmetic operations.

4. INTERVAL ARITHMETIC IN CUDA

Threads in CUDA are identifiable based on the previously mentioned indexing method. It provides a natural way to perform computation on data structures organised as a vector, matrix, or volume [11]. For model checking pIMCs, this allows

for a natural indexing method for evaluation of models with one to three parameters. It was found to be uncommon for parametric models to contain more than three parameters, thus by organising the parameter space in one of these data structures, it is possible to naturally integrate the model checking with CUDA parallelisation. To determine the initial parameter instantiation, the thread identifier formula in **Figure 3** can be utilised in combination with the step size desired for the intervals.

$$id_a = block.a \cdot blockDim.a + thread.a \qquad \text{for } a \in \{x, y, z\} \tag{8}$$

$$lower_a = id_a \cdot step \tag{9}$$

$$upper_a = id_a \cdot step + step \tag{10}$$

Figure 3. Thread identifier formula and bounds formulas.

When launching a kernel, the values for a maximum of three dimensions can be defined for both threads and blocks. The example in **Figure 4** represents a parametric model checking of a pIMC with two parameters, with the kernel being organised using 2x2 blocks and 3x3 threads per block. Each box in the grid represents a thread and contains its block thread index within. The parameter space is naturally split between the thread blocks, represented by the coloured regions, and indexed by the values to the left and below the grid. The thread indices within the blocks determines the final component for which interval a particular thread works on.

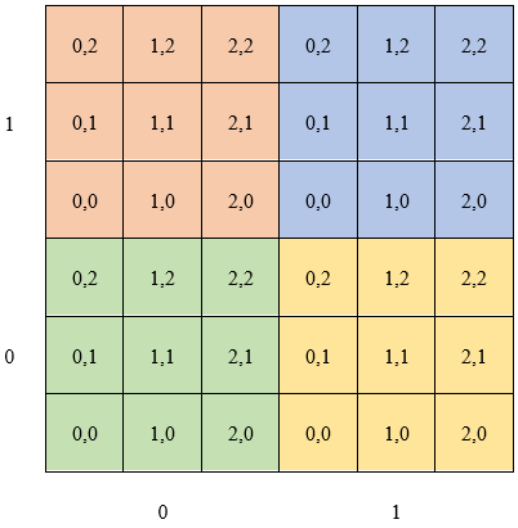

Figure 4. Thread block organisation in CUDA.

If one were to use a step size of 0.2, one would only require five threads in both the x- and y-axis, but this is not possible with the current number of thread blocks. Thus, when launching threads for a kernel, one may find that there is a surplus of threads, which would return immediately instead of performing the computations, but still cause a certain overhead due to the creation and destruction of these threads. Additionally, when deciding a thread block size, it is common to utilise a size which is a multiple of warp size to maximise concurrency [11].

Translation of the resulting thread identifier into a specified interval is trivial. The thread identifier and step size determine the lower bound of the parametric interval, and the step size in combination with the lower bound is used to find the upper bound. Additionally, as C and C++ use row-major order – or lexicographical access order - for the indexing of arrays [14], one can utilise the thread identifiers to determine the memory location where the result must be stored in a flattened array using the formulas found in **Figure 5** and the formula for id_a from **Figure 3**.

$$index = id_x \tag{11}$$

$$index = id_x \cdot n_y + id_y \tag{12}$$

$$index = id_x \cdot n_y + id_y \cdot n_z + id_z \tag{13}$$

Figure 5. Array index formula for 1, 2, or 3 parameters.

5. CUDA OPTIMISATIONS

CUDA programs can be optimised from a runtime and memory perspective by spreading the work evenly between the threads. This can be achieved by affecting the dimensionality of the kernel and its thread blocks. By purely trying to maximise the number of threads in each block, CUDA might not compile the program at all due to memory limitations enforced on each thread block. Thus, it is common practice for kernels to be launched with 256 threads per block, instead of the maximal 1024 [11]. The following sections will describe the optimisation efforts performed during this research, considering the compute capability of 6.1 of the GPU that experiments were performed on.

5.1 Kernel Dimensions

Regardless of compute capability, each block in CUDA can contain a maximum of 1024 threads. For a parametric model with two parameters, this equates to a thread block with a dimension of 32x32 threads per block. However, due to the large number of instructions performed during the evaluation of complex models, the kernel may have issues with launching due to memory limitations imposed on the thread blocks. Each thread block may utilise a maximum of 48KB shared memory, and each multiprocessor may utilise a maximum of 96KB of shared memory [11].

Lowering the number of threads per block and increasing the total number of blocks in the grid allowed for the spreading of shared memory more evenly throughout the multiprocessors, such that the kernel can launch and perform the parallelisation of the problem. For this research, the maximal block dimension possible for launching the kernel for each experiment was used to provide more consistent results between different parametric models.

5.2 Occupancy

Multiprocessor occupancy for CUDA programs refers to the ratio of concurrent warps versus the maximum number of warps supported on a multiprocessor of the GPU [11]. This value, represented by a percentage, is used to determine how many threads are active compared to how many threads could be active concurrently. For this research, the most major bottleneck for maximising occupancy was the register usage of the program. As the number of lines of code produced by the ePMC tool was found to range from a few thousand to over a million, most programs compiled with default options utilised 128 registers per thread.

NVIDIA has created a program called Nsight Compute [12], which can be utilised to simulate different kernel launches for calculating occupancy for specific hardware. For the GPU used in the experiments, occupancy could be maximised by having each thread utilise 32 registers on the device when using a kernel with 256 threads per block. The experiments section compares performance of CUDA programs compiled with both 128 and 32 registers, prioritising a balance between register and instruction optimization and occupancy respectively.

5.3 Grid-Stride Loops

Choosing which part of the problem each thread works on is another important consideration from the perspective of thread safety and memory usage. Grid-stride loops can be utilised to reduce the number of threads needed for computation. This technique makes each thread work on multiple inputs instead of a single interval. The benefit of this is the minimisation of the overhead for creating and destroying threads by spawning enough threads to saturate all cores of the GPU and perform all calculations for the specific problem using this maximal number of concurrent threads [7]. This research performed initial testing using grid-stride loops but found the results to be inconclusive. Thus, the experiments section does not contain the results of these tests, and the utilisation of this technique is left for future work.

5.4 Shared Memory

As memory in the GPU is a limited resource, the choice was made to calculate the intervals in-line within the kernel instead of storing each parameter combination before executing the kernel. Memory shared between the CPU and GPU was only reserved for storing the lower and upper bound results and then used to compare the results of the CPU and GPU implementations. As each thread can calculate its own index for storing the results of disjoint interval instantiations, there was no concern for thread safety when accessing these shared arrays.

6. EXPERIMENTS

Results of the parallelised CUDA program will now be compared against a single-threaded C++ implementation. Two parametric models from the PRISM benchmark suite were used for the experiments. They were conducted on a laptop with an Intel Core i7-10750H processor at 2.6GHz, an NVIDIA Quadro P620 graphics card, and a total of 16GB of RAM for the host, running on Ubuntu 20.04.

Each experiment compares the CUDA programs run-time for model checking the entire parameter space against a single threaded approach for evaluating the same. The programs were generated using the extended version of the ePMC tool by Gainer et al. [2018] and are compiled in a single CUDA program. For each combination of constants in the models, the CUDA program was ran using a step size of 0.001, block dimension of 63x63, and 16x16 threads per block. The tables also display information from the ePMC tool related to the size of the arithmetic circuit, which directly affects the number of instructions performed in both the CPU and GPU programs.

Additionally, both CUDA programs for the parametric models were compiled using 128 (GPU128) and 32 (GPU32) registers per thread. The latter was the highest possible register amount for the GPU to achieve 100% occupancy, while with a balanced optimisation level the compiler allocated 128 registers per thread, but only achieved 25% occupancy. In summary, the trade-off was four times the number of registers for four times higher occupancy. Regardless of execution on CPU or GPU, the result of the evaluation in each experiment was equal, not including edge cases involving results produced from models where division by 0 occurred. Results between the three approaches are compared in the tables in each subsection.

6.1 Bounded Retransmission Protocol

The bounded retransmission protocol (BRP) [8] is a variant of the alternating bit protocol. It divides a file into N chunks with each chunk allowed at most MAX retransmissions. The file is transmitted over two lossy channels K and L which send data and acknowledgements, respectively. The model is parametrised by pK and pL, which represents the reliability of each channel. Table 1 shows the performance statistics, measured in seconds, for the CPU and GPU execution times of each test case. The number of nodes in the DAG produced by the ePMC tool is displayed in the third column, which directly influences the number of instructions to be executed on both the CPU and GPU. The increase in performance is measured in the last column as a ratio of CPU and GPU128 execution times.

Table 1. Performance statistics for BRP.

N	MAX	Nodes	CPU	GPU128	GPU32	Incr.
64	4	15,367	105.67	0.76	0.75	137x
64	5	18,117	107.67	0.76	0.77	141x
256	4	61,639	418.75	2.82	2.84	148x
256	5	72,645	418.32	2.85	2.85	146x
512	4	123,335	839.76	5.57	5.61	150x
512	5	145,349	836.15	5.61	5.60	149x

The CUDA program was on average approximately 145 times faster than the single-threaded approach. The speedup was extremely consistent throughout each test and the run-time grew linearly in relation to N. In this experiment, the register limited kernel performed equally well in comparison to the higher register - but lower occupancy - kernel.

6.2 Crowds Protocol

The crowds protocol [13] models anonymity for Web browsing using M dishonest users who route their communications randomly through R different path reformulates within a group of N users. It is parametrised by $B = M / (M + N)$, the ratio of dishonest users compared to the total crowd size, and P, the probability of a member of the crowd sending a package to a randomly selected receiver. Table 2 shows the performance statistics, measured in seconds, for the CPU and GPU execution times of each test case.

Table 2. Performance statistics for crowds protocol.

N	R	Nodes	CPU	GPU128	GPU32	Incr.
5	3	3,495	3.96	0.08	0.07	38x
5	5	24,859	70.90	0.48	0.76	145x
5	7	105,359	254.76	1.81	3.22	140x
10	3	29,005	14.73	0.17	0.17	81x
10	5	465,928	346.91	2.36	6.03	146x
10	7	4,042,917	-C-	-C-	-C-	-C-
15	3	114,066	26.01	0.25	0.24	104x
15	5	3,265,023	-C-	-C-	-C-	-C-
15	7	-M-	-M-	-M-	-M-	-M-
20	3	314,925	37.30	0.32	0.31	117x
20	5	14,113,266	-C-	-C-	-C-	-C-
20	7	-M-	-M-	-M-	-M-	-M-

From the results of the crowds protocol experiments, it was found that the compilation and execution time increased exponentially depending on R, the amount of path reformulates. Cells marked as M denote memory limitations encountered during the ePMC function generation phase, while cells marked as C denote the compiler not managing to finish within a ten minute limit.

At best, the speed up was similar to the bounded retransmission protocol, and at worst the speedup was only a factor of 30 faster

in comparison to the single threaded approach. The average performance increase was approximately 110 times when comparing the CUDA program to the single-threaded evaluator. The register limited CUDA program performed noticeably worse in this experiment, most likely due to the high number of nodes in the DAG produced by the ePMC tool, and therefore a much longer kernel program.

7. CONCLUSIONS

In this paper a parallelised approach for the evaluation of parametric model checking of Markov Chains was implemented using CUDA. While instruction sets representing rational functions of stochastic models were extremely long, the kernels generated using this methodology could handle the evaluation of these parametric models and generated equal results compared to CPU implementations.

Parallelisation of the evaluation proved to be immensely faster than a purely single-threaded approach. Most test cases showed that parallelising evaluation can produce speedups of up to 140 times faster compared to sequential evaluation. Parametric model checking was found to fit naturally with the indexing methodology of CUDA and the value of parallelisation of parametric model checking was clear from the results of the experiments.

Several kernel dimensions were tested during the implementation of the evaluator. It was found that 16x16 threads per block was the maximal possible symmetric thread block to evaluate all experiments. The number of blocks in the grid was then evaluated based on this thread block size and the number of intervals requested for each parameter. Further optimisation of the kernel was left for future work due to time limitations.

In the future, this work could be expanded to utilise dynamic parallelisation for the minimisation of interval evaluation based on upper or lower bounds required for the result of the evaluation. Dynamic parallelisation is a technique in which child grids are launched within a parent grid based on specific conditions, namely result bounds in this case [1]. Additionally, the automatic generation of the CUDA program could be extended to split the extremely long kernels into multiple smaller kernels for the interest of compilation time and thus allowing for lower register usage to maximise occupancy. Following the compiler optimisations, optimal grid dimensions could be re-evaluated based on the naturally lower register and shared memory usage per kernel. Finally, the generated instruction set could be analysed and repeating instructions could be split into their own functions to maximise code reuse. As this parallelised approach was an extension of the previously created ePMC tool, any optimisations made in it such as improved reuse of repeating functions within the rational function representing the parametric model would be reflected in the compilation and execution times of this approach.

8. ACKNOWLEDGMENTS

I would like to thank my supervisor Moritz Hahn for his constant support and the authors of PRISM for their work on parametric models.

9. REFERENCES

[1] Andy Adinets, 2014. Adaptive Parallel Computation with CUDA Dynamic Parallelism. Retrieved from https://developer.nvidia.com/blog/introduction-cuda-dynamic-parallelism/.

[2] Anicet Bart, Benoit Delahaye, Didier Lime, Eric Monfroy, Charlotte Truchet. 2017. Reachability in Parametric Interval Markov Chains Using Constraints. arXiv:1706.00270. Retrieved from https://arxiv.org/abs/1706.00270.

[3] Dimitri P. Bertsekas and John N. Tsitsiklis. 2008. *Introduction to Probability* (2nd ed.). Athena Scientific, Belmont, Massachusetts.

[4] Hend Dawood. 2011. *Theories of Interval Arithmetic: Mathematical Foundations and Applications* (1st. ed.). LAP LAMBERT Academic Publishing, Saarbrücken, Germany.

[5] Paul Gainer, Ernst Moritz Hahn, and Sven Schewe. 2018. Accelerated Model Checking of Parametric Markov Chains. arXiv:1805.05672. Retrieved from https://arxiv.org/abs/1805.05672.

[6] Ernst Moritz Hahn, Tingting Han, and Lijun Zhang. 2011. Synthesis for PCTL in Parametric Markov Decision Processes. In *Lecture Notes in Computer Science*, *6617*, April 18-20, 2011, Pasadena California. Spring, Berlin, Heidelberg, 146–161. https://doi.org/10.1007/978-3-642-20398-5_12.

[7] Mark Harris, 2013. CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops. Retrieved from https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/.

[8] L. Helmink, M. Sellink and F. Vaandrager. 1994. Proof-checking a data link protocol. In *Proceedings of the international workshop on Types for proofs and programs (TYPES '93)*, Springer-Verlag, Berlin, Heidelberg, 127-165.

[9] Sebastian Junges, Joost-Piete Katoen, Guillermo A. Pérez, and Tobias Winkler. 2021. The complexity of reachability in parametric Markov decision processes. *Journal of Computer and System Sciences*, *119*, (Aug. 2021), 183–210. DOI: https://doi.org/10.1016/j.jcss.2021.02.006.

[10] D. Knuth and A. Yao. 1976. The complexity of nonuniform random number generation. In *Algorithms and Complexity: New Directions and Recent Results*, Academic Press, Inc., USA.

[11] NVIDIA. 2022. CUDA C++ Programming Guide. Retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[12] NVIDIA. 2022. NVIDIA Nsight Compute. Retrieved from https://developer.nvidia.com/nsight-compute.

[13] Michael K. Reiter and Aviel D. Rubin. 1998. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security* 1, 1 (Nov. 1998), 66-92. DOI: https://doi.org/10.1145/290163.290168.

[14] Peter S. Pacheco and Matthew Malensek. 2011. An Introduction to Parallel Programming (1st ed.), Morgan Kaufmann Publishers Inc., San Francisco, CA.

[15] Marc A. Suchard, Quangli Wang, Cliburn Chan, Jacob Frelinger, Andrew Cron, and Mike West. 2010. Understanding GPU Programming for Statistical Computation: Studies in Massively Parallel Massive Mixtures. *Journal of computational and graphical statistics: a joint publication of American Statistical Association, Institute of Mathematical Statistics, Interface Foundation of North America*, 19, 2, 419–438. DOI: https://doi.org/10.1198/jcgs.2010.10016.