Proceedings of the 2022 Winter Simulation Conference
B. Feng, G. Pedrielli, Y. Peng, S. Shashaani, E. Song, C.G. Corlu, L.H. Lee, E.P. Chew, T. Roeder, and P. Lendermann, eds.

GPU-ACCELERATED SIMULATION ENSEMBLES OF STOCHASTIC REACTION NETWORKS

Till Köster
Leon Herrmann
Philipp Andelfinger
Adelinde Uhrmacher

Institute for Visual and Analytic Computing
University of Rostock
Albert-Einstein-Str. 22
Rostock, 18059, GERMANY

ABSTRACT

Stochastic Simulation Algorithms are widely used for simulating reaction networks in cellular biology. Due to the stochastic nature of models and the large parameter spaces involved, many simulation runs are frequently needed. We approach the computational challenge by expanding the hardware used for execution by massively parallel graphical processing units (GPUs) to execute these ensembles of runs concurrently in a form of coarse-grained parallelization. Such computing infrastructure in the form of GPUs is readily available in desktop workstations and clusters but is not commonly exploited as part of stochastic simulation studies. Building on the existing literature in the field, we employ state-of-the-art algorithms to study the degree to which GPUs can augment the computation resources available for ensemble studies. Furthermore, the challenge of efficient work assignment given the GPU's synchronous mode of execution is explored. There are several algorithmic tradeoffs to consider for models with different execution characteristics, which we investigate in a performance study across four different models. To explore the limitations of the GPU-based simulators, the performance characteristics when executing large models are compared to those of highly optimized CPU simulators. Our results indicate that for some models adding a typical desktop GPU has a similar effect on performance as up to 40 added CPU cores.

1 INTRODUCTION

Alongside traditional wet-lab experiments, simulation has evolved to be a core branch of scientific work in the domain of cellular biology. One particularly successful class of models is that of (stochastic) reaction networks (Loskot et al. 2019). For this model type, the system's state consists of specific amounts of various species or reactants. Only the total amount of each species is considered, not their spatial distribution as an approximation of the *well-stirred system* (Gillespie 2007). The amounts are propagated through time using a set of reactions. Each reaction updates the state vector by decreasing the amount of some species and increasing the number of others. The likelihood of a reaction firing (or, more generally, the speed of the reaction) is determined by the propensity. This propensity is a function of the current state of the system. The most common way to calculate the propensity is through *mass action kinetics* where the propensity is proportional to the product of the amount of the individual reactants multiplied with a reaction constant. Reaction networks can be executed deterministically or stochastically. In the latter case, which we will focus on for this work, instead of mapping the reaction network into a set of coupled differential equations, a *stochastic simulation algorithm* (SSA) executes individual reactions stochastically as discrete events in continuous time, and updates time and state vector accordingly. Thereby, most of the SSAs (as does our approach) assume the time intervals between successive events to be exponentially distributed.

```
step():
  k = select_weighted_random(propensities)
  execute_reaction(k)
  for i in all_reactions:
    propensity[i] = calc_propensity(i)
```

(a) Direct method

```
step():
  k = select_weighted_random(propensities)
  execute_reaction(k)
  for i in dependent_reactions(k):
    propensity[i] = calc_propensity(i)
```

(b) Optimized method

Figure 1: Pseudocode excerpt for a simulation step in two stochastic simulation algorithms.

Stochastic simulations are computationally expensive, and due to their stochasticity require multiple replications. Most simulation experiments also rely on the variation of parameter configurations, e.g., simulation-based optimization, sensitivity analysis, or general parameter scanning (Kleijnen et al. 2005), increasing the required number of computations further. In this work, we will focus on the problem of replications, but the concepts introduced are also suitable for parameter studies.

The computationally challenging portion of SSA consists of two parts: randomly selecting a reaction to fire, weighted by its computed propensity, and updating the state and its propensities. The Direct Method (DM) (Gillespie 1977) (and its equivalent formulation, the First Reaction Method (FRM) (Gillespie 1976)) compute all propensities for all reactions at every step. A far more efficient approach for larger systems is to use a dependency graph to only update those propensities that could have changed. The difference in pseudocode of the two approaches is shown in Fig. 1. This dependency-based approach is used in the Optimized Direct Method (ODM) (Cao et al. 2004) for the DM and in the Next Reaction Method (NRM) (Gibson and Bruck 2000) for the FRM.

In addition to these algorithmic improvements, parallelization is another option to increase simulation performance (Fujimoto 2016). However, the steps in SSA are often tightly coupled as each executed reaction may change the propensities of many other possible reactions. Since each step is relatively fast (less than a microsecond), efficient fine-grained parallelization (i.e., of a single replication) is challenging (Dematté and Prandi 2010). As many replications have to be executed, a coarse-grained parallelization across large numbers of replications (also called ensembles), provides another viable option to increase the performance of SSA simulations, also on the GPU (Li and Petzold 2010).

Code generation is an established technique to generate an executable with minimal overhead from generic functionality (Futamura 1999). By providing the compiler with both the simulation algorithm and the specific model, compiler optimizations can consider the simulation as a whole. Code generation works particularly well in the context of SSA, where a large part of the computational load lies in computing model-specific (propensity) expressions (Köster et al. 2022).

Our aim is to study the extent to which GPU implementations following state-of-the-art principles can augment the computational power provided by traditional CPU-based implementations in large ensemble studies. To this end, we present detailed performance measurements of three variants of SSA on GPUs across models with different execution characteristics and scales. Our simulator implementations, data and experiment scripts are publicly available (https://git.informatik.uni-rostock.de/mosi/gpu-ssa-ensemble).

2 BACKGROUND AND RELATED WORK

In the following, we introduce the hardware characteristics of GPUs as well as the existing work on GPU-based acceleration of ensembles of SSA simulations.

2.1 General-Purpose Computing on GPUs

GPUs have evolved from fixed-function hardware targeting the rendering of three-dimensional scenes to general-purpose accelerators widely available in machines ranging from laptops to supercomputers. Common GPU frameworks such as OpenCL (Khronos OpenCL Working Group, Beaverton, OR, USA

2011) and NVIDIA CUDA (Cook 2012) allow developers to specify GPU programs, referred to as *kernels*, on the level of individual GPU *threads*. Threads are grouped into sets of a hardware-specific size of 32 or 64 threads referred to as warps or wavefronts. Within a warp, threads operate in lockstep. Hence, if the control flow of the threads within a warp diverges, all branches are taken in sequence while discarding unneeded results. In addition, if adjacent threads access adjacent locations in memory, the accesses are *coalesced* into a single memory transaction served as one.

Due to the architectural focus on executing thousands of arithmetic operations in parallel rather than on control flow and caches, algorithms involving regular control flow and memory access patterns are a natural fit for GPU-based parallelization. Although simulations often involve sparse and irregular computations and thus require GPU-specific adaptations and optimizations, simulations from various domains, including cell biology, have been shown to benefit immensely from GPU-based parallelization given suitable choices of algorithms and data structures (Park and Fishwick 2010; Zhu et al. 2011; Xiao et al. 2019).

2.2 SSA on GPUs

The parallelization strategies for GPU-accelerated SSA simulations found in the literature can be divided into two groups. In coarse-grained parallelization, multiple *entire simulations* are executed in parallel. Fine-grained parallelization, on the other hand, also parallelizes *individual simulations*, in addition to executing multiple simulations concurrently, for instance in (Sumiyoshi et al. 2015). In the Direct Method, for example, a fine-grained approach may parallelize the updates of the population counts and propensities. The choice between coarse-grained or fine-grained parallelization determines the amount of GPU resources that can be dedicated to each individual simulation run. Fine-grained approaches can utilize many parallel threads to execute a single run. Hence, coarse-grained parallelization, which is the prevalent method in the literature, focuses on supporting particularly large numbers of replications of comparatively small model instances.

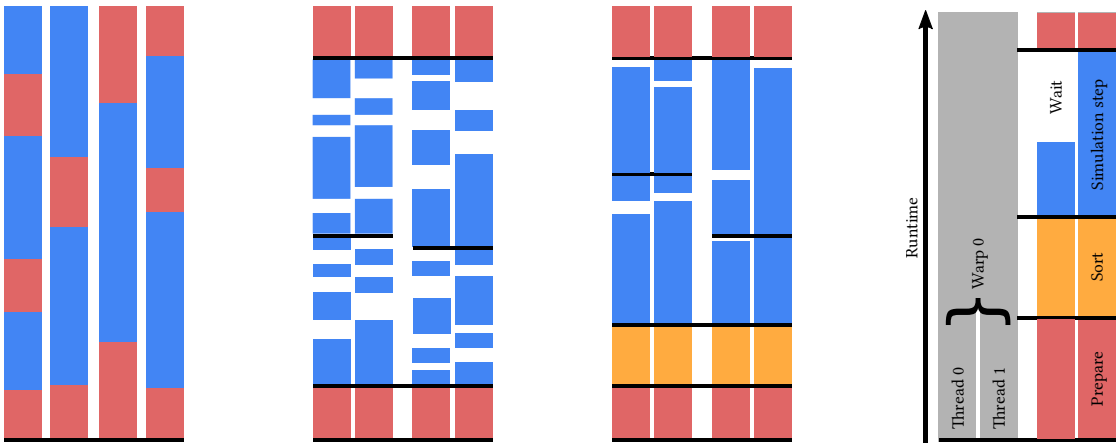
The Direct Method is used in these implementations (Jenkins and Peterson 2010; Klingbeil et al. 2012; Sumiyoshi et al. 2015), as its relatively simple control flow makes it reasonably straightforward to map to GPU execution. The simulator in (Li and Petzold 2010) uses the Logarithmic Direct Method. Dependency graph-based methods perform better for larger models, but are more difficult to implement and less prevalent in the literature. In (Klingbeil et al. 2011), the Next Reaction Method and the Logarithmic Direct Method are implemented in addition to the Direct Method. The closest existing work to ours presents an implementation of the Optimized Direct Method (Komarov and D'Souza 2012). To enable additional fine-grained parallelization, propensity values and their respective sums are updated in parallel using a prefix sum algorithm.

GPUs achieve highest performance when executing programs with largely homogeneous control flow and memory access patterns, which runs counter to the sparsity and heterogeneity of the computations involved when executing various parametrizations and replications of stochastic simulations in parallel. Thus, GPU-based SSA implementations require careful consideration of the GPU's execution model and hardware properties. A frequently employed means of optimizing execution speed is the efficient usage of the GPU's memory hierarchy. For instance, (Klingbeil et al. 2012) analyzes different data structures to store model parametrizations in fast read-only regions of GPU memory. Consideration is also given to dense and compressed storage of data (Komarov and D'Souza 2012). In our work, we generate specialized simulators using partial evaluation via code generation (Futamura 1999), in which full knowledge of the considered model is available during compilation, allowing the compiler to generate optimized model-specific machine code. We previously demonstrated the benefits of code generation for CPU-based SSA (Köster et al. 2022). In the GPU context, code generation avoids the need for individual threads to traverse complex data structures at simulation runtime and thus reduces the need for manual optimization of memory access patterns.

The issue of thread divergence can be addressed by sorting the computations involved in different simulation replications so that adjacent threads execute the same control flow (Kunz et al. 2012). We

present detailed measurements assessing the performance benefits and overhead when sorting impending reactions of SSA replications.

3 OUR APPROACH



(a) CPU (b) GPU, no sorting (c) GPU, sorting (d) Legend

Figure 2: An illustration of the methods we use for efficient execution. On the CPU, different threads do not interact and run independently (2a). When running on the GPU, different simulation steps of the different replications in the ensemble diverge, leading to serialization (2b). An added initial sorting step reduces divergence during execution (2c).

This section will give an overview of our approach to executing ensembles of SSA runs to GPU execution, noting where we differ from and expand the state of the art. In contrast to most of the related work, our approach uses the Optimized Direct Method (ODM) (Cao et al. 2004) in conjunction with code generation. The ODM has known good scaling behavior. The used data structures are simpler than in the NRM and thus lend themselves more to mapping to GPUs.

We considered three different approaches, all of which use coarse-grained parallelization, i.e., parallelization across multiple simulations. In preliminary experiments with a fine-grained parallelization, we observed no performance benefits for the models considered in this paper.

3.1 Code Generation

Traditionally, SSA simulators adhere to a strict separation between the simulator and the model comprised of production rules and initial populations. In this approach, generic simulator procedures handle functionalities such as the updating of propensities during a reaction, with dynamic concrete control flow and memory accesses according to the model data. We have previously shown that the performance of CPU-based SSA simulations can benefit immensely from code generation (Köster et al. 2020), which produces specialized simulation code combining a simulator and a specific model. Code generation offers the opportunity for compiler optimizations based on full knowledge of the simulator and the model. For instance, knowledge of the dependencies among the reactions reveals the number of propensity updates per reaction type and the affected propensities. In the GPU context, static optimizations of the memory access patterns may increase the opportunities for memory access coalescing, at the potential cost of increased divergence given the separation of update procedures by reaction type.

3.2 Propensity Sorting

The selection of the next reaction to occur involves a weighted sampling from the list of propensities. By sorting the propensities based on values gathered from previous simulation runs, as originally described in the context of the ODM (Cao et al. 2004), more efficient sampling can be achieved. The goal of the propensity sorting is to speed up the linear reaction selection by moving frequently firing reaction to the beginning of the array to be searched. The sorting occurs only once as a preprocessing step. Initial simulation runs are executed to count how often each reaction fires in the course of a simulation. The propensity list is then sorted using these reaction counts as estimates for future simulations.

3.3 Computation Sorting

As a first approach, we implemented an ODM simulation step in a single kernel that updates the populations and propensities affected by a reaction. ODM employs a dependency graph to avoid unnecessary updates. Thus, in a given simulation step, the different replications may execute different reactions which causes GPU threads within a warp to diverge both in the execution and the update step (Fig. 1b). The resulting serialization within warps is illustrated in Figure 2b. In this first approach, we do not mitigate the effects of divergence.

This can be contrasted with the Direct Method, which recalculates all propensities (Fig. 1a) after every reaction and thus does not scale well to large reaction counts. However, since the Direct Method lends itself to straightforward implementation targeting GPUs, we will use it as a baseline in our experiments.

Our second approach aims to minimize divergence by sorting the replications by the reaction they choose in each simulation step. The basic simulation algorithm and its implementation remains unchanged. However, simulation replications are allocated to the threads in a sorted fashion according to the selected reactions. This increases the likelihood that identical reactions occur in adjacent threads, reducing divergence. The sorting introduces additional computational as well as memory access overhead. It should, however, decrease thread divergence, because most consecutive threads execute the same reactions, as long as the number of possible reactions in the system is significantly smaller than the number of threads times the number of replications. Figure 2c visualizes how the sorting takes additional execution time, but increases the effective parallelism. For contrast, we also show the execution on a CPU (cf. Fig. 2a): since CPU threads work largely independently of each other, the individual simulations can progress independently.

We implemented two different variants of sorting. In the *logical sorting* method, only an index map is sorted based on the next reaction indices. The simulation variables are then accessed according to the sorted map so that most consecutive threads execute the same reaction. The *physical sorting* method, on the other hand, rearranges all simulation state variables based on the chosen next reactions in memory. The main difference between these sorting methods lies in the resulting access patterns to the simulation state memory. Since only an index map is sorted within the logical sorting method, the memory accesses made by adjacent threads are still scattered to different portions of the simulation state, limiting the opportunities for memory access coalescing (cf. Section 2.2).

A more favorable memory access pattern is achieved by the physical sorting method, in which the n-th thread processes the simulation variables at the n-th index. On the other hand, the physical sorting method incurs a higher memory access overhead during the sorting step.

3.4 Implementation

Our GPU simulators are written in C++ and use NVIDIA CUDA together with the high-level library Thrust (Bell et al. 2017), which provides data parallel primitives such as scans, sorts and reductions. The simulator code and our plotting scripts are available at https://git.informatik.uni-rostock.de/mosi/gpu-ssa-ensemble.

We reused the model parsing source code from our previous work targeting CPUs (Köster et al. 2020) but created a new GPU compatible simulator template. Code generation comes with a higher one-time cost of compiling the generated code, but in our scenario of interest, this is of minor concern, as we are interested

Table 1: Overview of the evaluated models, together with their respective amounts of species and reactions. The model degree describes the average number of propensities (based on the dependency graph) that must be updated after executing a single model reaction.

Model	Species	Reactions	Degree
SIR with N regions	3N	~4N	~6
Decay dimerization	3	4	3
Multistate	9	18	7
Multisite	66	288	55

in many replications where these one-time costs amortize. Compilation times for small to medium-sized models were similar to a larger CUDA program (On the order of 10 to 100 seconds for the models tested).

The next reaction indices for each replication are chosen through linear weighted random sampling. An alternative would have been a binary search as described for the LDM (Li et al. 2006). We chose the linear search for our implementation because it was originally described for the ODM (Cao et al. 2004), and in our experience works well for many models of practical relevance when used in conjunction with propensity sorting. For comparability, we use the linear search mode in both the GPU and the CPU simulators. The ODM uses two random numbers in each simulation step for each replication. Those random numbers are generated using the CUDA's default XORWOW pseudo-random generator.

Our implementation deviates from the original descriptions from our CPU-based simulator (Köster et al. 2022) with respect to the updates of the sum of all propensities, which is required in each simulation step. While those updated the propensity sum incrementally based on the change in the current step, our preliminary experiments showed that in the GPU context, it is typically more efficient to sum all propensities in each step instead.

4 EXPERIMENTS

We consider four models to test the suitability of the different simulators and to obtain a comparison to the sequential implementation. Table 1 gives an overview of the models. To provide a comprehensive comparison to a CPU-based execution, we include model instances larger than typically encountered in the literature, where models with less than 10 reactions are prevalent.

The **SIR (Susceptible, Infected, Recovered) model with regions** is a variant of the classic SIR model (Kendall 1956). The regular SIR model describes the course of a disease in a closed population. Viewed as a reaction network, the model includes three species – Susceptible, Infected and Recovered – which can react in the following two ways:

$$S + I \to I + I \qquad I \to R \ .$$

The first reaction describes an infection and the second one a recovery. The SIR model with regions arranges multiple of these SIR models in a circle. An example of our SIR region model with four regions is visualized in Figure 3. The simulation is initialized with a single infected individual present in the overall system.

The **decay dimerization model** (Gillespie 2001) is based on the following four reactions that describe the possible interactions between three species:

$$S_1 \to \emptyset \qquad S_1 + S_1 \to S_2 \qquad S_2 \to S_1 + S_1 \qquad S_2 \to S_3 \ .$$

We parametrized the model using the reaction rates and initial populations from (Gillespie 2001). All simulations were executed up to a simulation time of 10. Simulation time here is the virtual time within the model, measured in the model's internal unit system.

The **Multistate and the Multisite model** (Blinov et al. 2004) represent typical biochemical networks. Both were designed for the illustration of biochemical reaction networks and deal with the phosphorylation

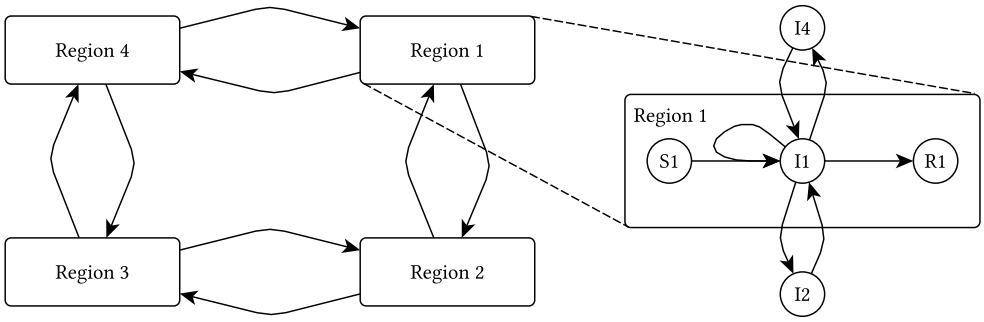


Figure 3: Structure of the SIR model with four regions.

as either different states of the molecules or at different sites of the molecules. They were simulated up to a time of 10 and 4 respectively. The models have been used previously in a performance review (Gupta and Mendes 2018) as well as for the evaluation of the CPU simulator (Köster et al. 2022) used as a baseline in our performance evaluation.

4.1 Reference Simulators and Environment

We compare the performance of our GPU implementations to our CPU-based simulator (Köster et al. 2020). This simulator implements a variant of the ODM and uses code generation in combination with partial evaluation to create specialized simulators for different stochastic models. We previously showed that for the Multistate and the Multisite model, this CPU-based simulator outperforms both the simulator integrated into BioNetGen as well as the simulators evaluated in a recent performance review (Gupta and Mendes 2018; Köster et al. 2022).

In addition to our GPU simulator variants based on ODM, we implemented a baseline GPU simulator using the original Direct Method without reaction sorting and code generation. While we expect the Direct Method's performance to be comparatively low for large models, this simulator variant provides an indication of the performance of a basic SSA implementation on a GPU. The experiments with this simulator serve to gain a better understanding on how the use of ODM, computation sorting, and code generation affects the performance. The simulations were run on a Windows 10 system with an i7-8700K Intel CPU with frequency scaling enabled, and an NVIDIA GeForce GTX 1070 GPU. The CPU simulator was evaluated running in Debian 10.5 on the Windows Subsystem for Linux (WSL2). This simulator was compiled using gcc version 8.3.0. The CPU simulator was executed using a single CPU thread. The C++ compiler version for the GPU simulators is 19.28.29912.0 (Visual Studio 16 2019) with CUDA version 10.2.

4.2 Results

In Figure 4, we give an overview of the different methods compared in our performance evaluation. The results of the performance evaluation can be found in Figure 5. As a performance metric, we use the wall-clock execution time in μs per step, which makes simulation results easily comparable across simulators and models. Five simulation runs consisting of many individual replications were executed to calculate the average performance values for the GPU simulators. The CPU performance is calculated as the average of ten measurements with 100 replications. All performance values are measured without code generation, compilation, or initialization times. The initialization time includes initial memory allocation as well as the transfer of initial simulation states to graphics memory. These costs amortize when running many replications. To better understand the impact of sorting, we measured the time taken to sort explicitly.

To verify correctness, we configured the GPU simulators to output the entire system trajectory, which involves substantial overhead. However, in real-world applications, the number of relevant observables

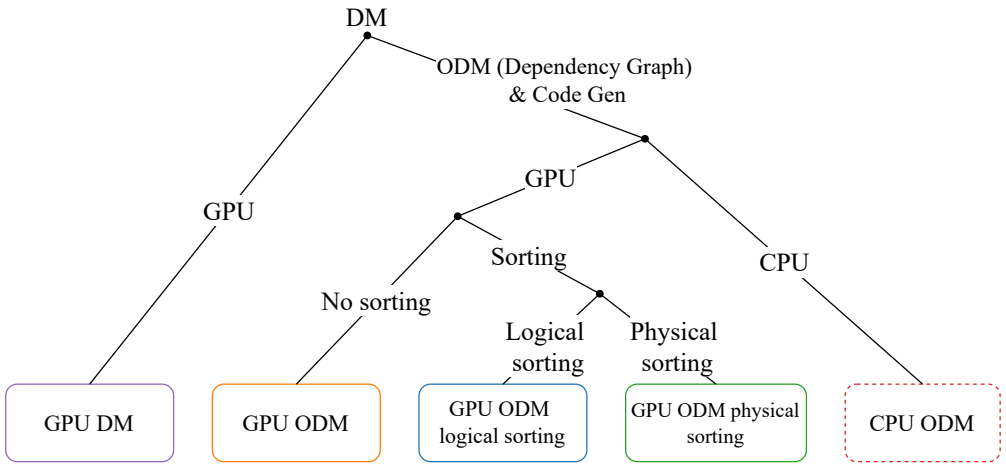


Figure 4: Hierarchy and relation of methods used for evaluation. Colors correspond to those used in Fig. 5.

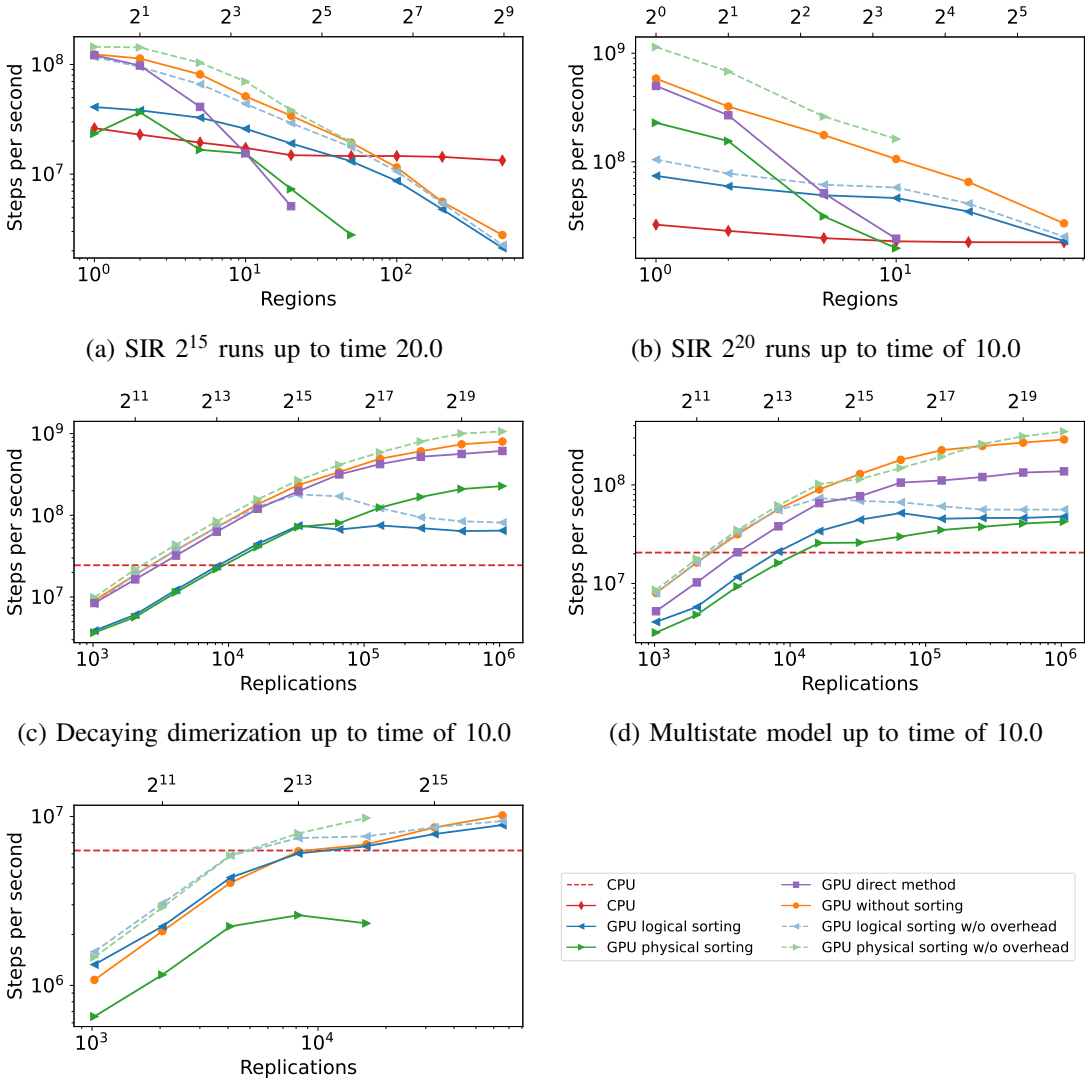
can be assumed to be small, introducing only little overhead. We therefore performed the measurements without output or storage of observations.

4.2.1 SIR Model with Regions

Figures 5a and 5b show the results for the performance measurements of two different model configurations of the SIR model with regions. The results shown in Fig. 5a originate from simulations with 2^{15} $(= 32\,768)$ replications and with a maximum simulation time of 20. Fig. 5b on the other hand, is based on simulations with 2^{20} $(\approx 10^6)$ replications which run up to a simulation time of 10. The results show that the performance of all simulators decreases with an increasing number of regions, which is to be expected as the reaction selection becomes more computationally expensive with larger models. Apart from the possibility of increased thread divergence due to larger numbers of reactions, the cost of the population and propensity updates should be unaffected by different sizes of the SIR region model, since each reaction updates two populations and at most six propensities.

However, in our GPU implementation, the cost of updating of the total propensity varies with different model sizes as all individual propensities are traversed. Compared to any of the GPU simulators, the performance of the sequential simulator is not as strongly affected by the varying model sizes. This can be seen both in Figures 5a and 5b. We attribute the more significant influence of the model size on the GPU simulators to the reaction selection and the recalculation of the total propensity. Linearly selecting the next reactions for all threads within a warp delays the progress of all threads until the slowest selection has completed. When the model instances grow in size and therefore have more possible reactions, it becomes more likely that one thread takes longer to choose a reaction and in turn, slows down all threads in the same warp. In addition, our implementation of the recalculation of the total propensities has to iterate over all propensities and consequently slows down for instances with more reactions.

Overall, the non-sorting method has the highest execution speed. The CPU simulator performs best only for large instances with relatively few replications. The GPU Direct Method performs well for small instances of the SIR region model, but slows down with larger instances, because it lacks the optimizations of the other simulators. Both sorting methods perform worse than the non-sorting method for this model. To understand the trade-off between the sorting overhead and its benefit during the execution of a simulation step, we also show results when subtracting the time spent on sorting. When disregarding the sorting overhead, the execution speed of the physical sorting method exceeds that of the non-sorting method. This shows that the additional sorting step has the potential to speed up the execution of the simulation steps. Still, the overhead introduced by this method far outweighs the performance gain.



(a) SIR 2^{15} runs up to time 20.0

(b) SIR 2^{20} runs up to time of 10.0

(c) Decaying dimerization up to time of 10.0

(d) Multistate model up to time of 10.0

(e) Multisite model up to time of 4.0

Figure 5: Performance graphs for different models and simulators.

The logical sorting method has a lower memory access overhead since only the index map is sorted. In contrast to the physical sorting method, subtracting the sorting overhead does not improve the performance significantly. For low numbers of replications (Figure 5a), the performance without the overhead is comparable to the non-sorting method. However, sorting only the indices when simulating many replications (Figure 5b) lowers the performance significantly. This is because rearranging the indices results in a non-optimal memory access pattern for the replication variables, which in turn provides fewer opportunities for memory access coalescing. This effect worsens with more replications.

4.2.2 Decay Dimerization Model

The Figure 5c shows that the execution speed of the decay dimerization model on a GPU increases with the number of replications. Overall, the execution speeds are higher than those of the SIR model, even though the regular SIR model with only a single region has only two reactions and a model degree of two,

compared to four decay dimerization reactions and a degree of three. This is because the SIR model may terminate after just one simulation step if the single initially infected entity heals before infecting other entities. The large variance in the simulation durations lowers the utilization of the GPU's resources. In contrast, the simulation durations of the decay dimerization model are more consistent, allowing for higher GPU utilization.

Similar to the SIR model results, the performance of the GPU simulator without sorting increases with the number of replications. Since the dimerization model is comparatively small and simple, the execution speed of the GPU Direct Method is roughly within 20% of the non-sorting ODM across all tested numbers of replications. Both sorting ODM simulators have similar execution characteristics up to about 2^{16} $(= 65\,536)$ replications. Beyond that point, the performance of the physical sorting method increases further, whereas the speed of the logical sorting method stabilizes. When comparing the results to those of the SIR model, the performance of the physical sorting ODM is approximately equal or greater than that of the non-sorting ODM, but only when excluding the sorting overhead. The performance of the non-sorting GPU simulator, as well as the GPU Direct Method, scales almost linearly with the number of replications up to a certain number of replications, beyond which the experiment approaches full saturation of the GPU resources. The performance of the GPU simulator without sorting increases linearly up to about 2^{14} $(= 16\,384)$ replications and saturates at a value of about 952 million simulation steps per second (1.05 ns per step). This throughput is the highest one of all simulators and models that were evaluated as part of this work. That means that the individual steps in this model are the cheapest to calculate compared to the other models. The sequential simulator reached an average execution time of about 24.5 million steps per second. The overall maximum speedup of the non-sorting GPU implementation is therefore about 39.

4.2.3 Multistate and Multisite Model

For the Multistate model, we find that the non-sorting method has, similar to the previous measurements, the highest overall execution speed when simulating large numbers of replications. Both sorting methods behave similarly when increasing the replication count. However, the sorting overhead of the physical sorting simulator is significantly higher than for the dimerization model because the Multistate model has more species, as well as more propensities to be sorted.

In general, the numbers of simulation steps per second are lower for all GPU simulators compared to the decay dimerization model. This can be attributed to the higher complexity, due to the larger number of reactions, and the higher numbers of species and reactions of the Multistate model. The Multistate model has 288 different reactions that update 55 propensities on average. We also observe an earlier diminishing of performance gains with more replications. The performance of the CPU simulator, on the other hand, is barely affected by the different structure of the Multistate model at 20.6 million steps per second compared to 24.5 million simulation steps per second for the dimerization model.

For the Multisite model, simulations were run up to a maximum simulation time of 4. The data shows that the implemented GPU methods are not well suited for this model. The performance of the CPU simulator running on two CPU cores would be better than that of all GPU simulators for all tested numbers of replications. One notable result, however, is that for this model, the logical sorting performs slightly better than the non-sorting approach for replication counts up to about 2^{12} $(= 4\,096)$. The higher performance of the sorting approach can be attributed to the smaller computational and memory access overhead of the sorting step for fewer replications and its performance improvements due to minimizing branching for the propensity and population updates. Due to the large number of reactions and propensities to be updated in the Multisite model, the runtime overhead for the reaction selection and population and propensity updates caused by branching is higher than for the previously mentioned models. Measurements for the GPU Direct Method are not included in Fig. 5e, because of its poor performance for large and complex model instances.

5 CONCLUSION

This paper presented a detailed performance comparison of three variants of the Optimized Direct Method for simulating large ensembles of reaction networks on a GPU. A state-of-the-art CPU implementation and a basic GPU implementation served as baselines. The simulators were evaluated on the example of three models from the literature across a wide range of replication counts. Generally, we observed that full utilization of a modern GPU's computational resources is achieved only at large replication counts between 100 and 1000, depending on model size. In the best case, a GPU provided the equivalent of around 40 CPU cores executing an optimized CPU implementation.

In addition to the use of code generation, we introduced a sorting step to make a more homogeneous control flow across simulations. We find that for some problems, the sorting reduces the execution time of the simulation step substantially. However, this speedup does not outweigh the overhead introduced by the sorting. Future work could explore a localized sorting that would reduce control flow divergence while inducing lower overhead. A basic GPU implementation of the Direct Method performed well for very small model instances but did not scale with increasing instance size. An attractive avenue for future work is the exploration of improved memory layouts using thread-local storage, which exhibited promising results in our initial experiments.

ACKNOWLEDGEMENTS

Financial support was provided by the Deutsche Forschungsgemeinschaft (DFG) research grant ESCEMMO (UH-66/13).

REFERENCES

Bell, N., J. Hoberock, and C. Rodrigues. 2017. "THRUST: A Productivity-oriented Library for CUDA". In *Programming Massively Parallel Processors*, 475–491. Elsevier, Amsterdam.

Blinov, M. L., J. R. Faeder, B. Goldstein, and W. S. Hlavacek. 2004. "Bionetgen: Software for Rule-based Modeling of Signal Transduction Based on the Interactions of Molecular Domains". *Bioinformatics* 20(17):3289–3291.

Cao, Y., H. Li, and L. Petzold. 2004. "Efficient Formulation of the Stochastic Simulation Algorithm for Chemically Reacting Systems". *The Journal of Chemical Physics* 121(9):4059–4067.

Cook, S. 2012. *Cuda Programming: A Developer's Guide to Parallel Computing with Gpus*. 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Dematté, L., and D. Prandi. 2010. "Gpu Computing for Systems Biology". *Briefings in bioinformatics* 11(3):323–333.

Fujimoto, R. M. 2016. "Research Challenges in Parallel and Distributed Simulation". *ACM Transactions on Modeling and Computer Simulation* 26(4):1–29.

Futamura, Y. 1999. "Partial Evaluation of Computation Process – an Approach to a Compiler-compiler". *Higher Order Symbolic Computation* 12(4):381–391.

Gibson, M. A., and J. Bruck. 2000. "Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels". *The Journal of Physical Chemistry A* 104(9):1876–1889.

Gillespie, D. T. 1976. "A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Reactions". *Journal of Computational Physics* 22(4):403–434.

Gillespie, D. T. 1977. "Exact Stochastic Simulation of Coupled Chemical Reactions". *The Journal of Physical Chemistry* 81(25):2340–2361.

Gillespie, D. T. 2001. "Approximate Accelerated Stochastic Simulation of Chemically Reacting Systems". *The Journal of Chemical Physics* 115(4):1716–1733.

Gillespie, D. T. 2007. "Stochastic Simulation of Chemical Kinetics". *Annual Review of Physical Chemistry* 58(1):35–55.

Gupta, A., and P. Mendes. 2018. "An Overview of Network-Based and -Free Approaches for Stochastic Simulation of Biochemical Systems". *Computation*.

Jenkins, D., and G. Peterson. 2010. "GPU Accelerated Stochastic Simulation".

Kendall, D. G. 1956. "Deterministic and Stochastic Epidemics in Closed Populations". In *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability, Volume 4: Contributions to Biology and Problems of Health*, 149–165. University of California Press.

Khronos OpenCL Working Group, Beaverton, OR, USA 2011. *The Opencl Specification, Version 1.1*.

Kleijnen, J. P., S. M. Sanchez, T. W. Lucas, and T. M. Cioppa. 2005. "State-of-the-art Review: A User's Guide to the Brave New World of Designing Simulation Experiments". *INFORMS Journal on Computing* 17(3):263–289.

Klingbeil, G., R. Erban, M. Giles, and P. K. Maini. 2011. "STOCHSIMGPU: Parallel Stochastic Simulation for the Systems Biology Toolbox 2 for MATLAB". *Bioinformatics* 27(8):1170–1171.

Klingbeil, G., R. Erban, M. Giles, and P. K. Maini. 2012. "Fat Versus Thin Threading Approach on GPUs: Application to Stochastic Simulation of Chemical Reactions". *IEEE Transactions on Parallel and Distributed Systems* 23(2):280–287.

Komarov, I., and R. M. D'Souza. 2012. "Accelerating the Gillespie Exact Stochastic Simulation Algorithm Using Hybrid Parallel Execution on Graphics Processing Units". *PLOS ONE* 7(11).

Köster, T., T. Warnke, and A. M. Uhrmacher. 2022. "Generating Fast Specialized Simulators for Stochastic Reaction Networks Via Partial Evaluation". *ACM Transactions on Modeling and Computer Simulation* 32(2).

Kunz, G., D. Schemmel, J. Gross, and K. Wehrle. 2012. "Multi-level Parallelism for Time- and Cost-efficient Parallel Discrete Event Simulation on Gpus". In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, 23–32: IEEE.

Köster, T., T. Warnke, and A. M. Uhrmacher. 2020. "Partial Evaluation Via Code Generation for Static Stochastic Reaction Network Models". In *Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '20, 159–170. New York, NY, USA: Association for Computing Machinery.

Li, H., and L. Petzold. 2010. "Efficient Parallelization of the Stochastic Simulation Algorithm for Chemically Reacting Systems on the Graphics Processing Unit". *The International Journal of High Performance Computing Applications* 24(2):107–116.

Li, H., L. Petzold, H. Li, and L. Petzold. 2006. "Logarithmic Direct Method for Discrete Stochastic Simulation of Chemically Reacting Systems". *Journal of Chemical Physics* 16:1–11.

Loskot, P., K. Atitey, and L. Mihaylova. 2019. "Comprehensive Review of Models and Methods for Inferences in Bio-chemical Reaction Networks". *Frontiers in Genetics* 10.

Park, H., and P. A. Fishwick. 2010. "A Gpu-based Application Framework Supporting Fast Discrete-event Simulation". *Simulation* 86(10):613–628.

Sumiyoshi, K., K. Hirata, N. Hiroi, and A. Funahashi. 2015. "Acceleration of Discrete Stochastic Biochemical Simulation Using GPGPU". *Frontiers in Physiology* 6.

Xiao, J., P. Andelfinger, D. Eckhoff, W. Cai, and A. Knoll. 2019. "A Survey on Agent-based Simulation Using Hardware Accelerators". *ACM Computing Surveys* 51(6):1–35.

Zhu, Y., B. Wang, and Y. Deng. 2011. "Massively Parallel Logic Simulation with Gpus". *ACM Transactions on Design Automation of Electronic Systems* 16(3):1–20.

AUTHOR BIOGRAPHIES

TILL KÖSTER is a doctoral researcher in the modeling and simulation group at the Institute for Visual and Analytic Computing, University of Rostock. His e-mail address is till.koester@uni-rostock.de.

LEON HERRMANN is a student in the Computer Science Master's program at the University of Rostock. His e-mail address is leon.herrmann@uni-rostock.de.

PHILIPP ANDELFINGER is a postdoctoral researcher in the modeling and simulation group at the Institute for Visual and Analytic Computing, University of Rostock. His e-mail address is philipp.andelfinger@uni-rostock.de.

ADELINDE M. UHRMACHER is a professor at the Institute for Visual and Analytic Computing, University of Rostock, and head of the modeling and simulation group. Her e-mail address is adelinde.uhrmacher@uni-rostock.de.