Accelerating the Smoldyn Spatial Stochastic Biochemical Reaction Network Simulator Using GPUs

Denis V. Gladkov, Samuel Alberts, Roshan M. D'Souza
Dept. of Mechanical Engineering
UW-Milwaukee
3200 N Cramer Street
Milwaukee, WI, 53211

Steven Andrews
Fred Hutchinson Cancer Research Center
1100 Fairview Ave. N. A2-025
P. O. Box 19024
Seattle, WA 98109

Keywords: Example keywords: Biochemical Reaction Networks, Smoldyn, GPGPU

Abstract

Smoldyn is a spatio-temporal biochemical reaction network simulator. It belongs to a class of methods called *particle-based methods* and is capable of handling effects such as molecular crowding. Individual molecules are modelled as point objects that can diffuse and react in a control volume. Since each molecule has to be simulated individually, the computational complexity of the simulator is quite high. Efficiently executing high fidelity models ($> 10^6$ molecules) is not feasible with traditional serial computing on central processing units (CPUs). In this paper we present novel data-parallel algorithms designed to execute on graphics processing units (GPUs) to handle the computational complexity. Our preliminary implementation can handle diffusion, zero-order, uni-molecular, and bi-molecular reactions. Our preliminary results show performance gain of over 200x over the original implementation without loss of accuracy.

1. INTRODUCTION

Many current questions in cell biology cannot be explored by qualitative reasoning alone and require quantitative computer models. These models usually focus on one or more portions of the intracellular chemical reaction network, which performs metabolism, signaling, development, cell division, and other vital cellular tasks. In the simplest models, which are based on ordinary differential equations, the intracellular volume is treated as a single well-mixed compartment and all chemical concentrations are treated deterministically [1]. While these simple models are convenient and widely used, they are also very limiting. They are inadequate for studying spatial aspects of intracellular processes, such as the diffusion of signaling proteins across a cell, and for studying stochastic aspects, such as the variability of protein concentrations that naturally arises from gene expression. They are also inadequate for modeling system behaviors that are affected by underlying spatial and stochastic chemical reaction dynamics [3]. To address the need for greater modeling detail, many simulators explicitly represent spatial and/or stochastic dynamics.

Three classes of biochemical simulators account for spatial and stochastic detail [16]: (*i*) *microscopic lattice simulators* represent space with a fine mesh and allow up to one molecule in each lattice site, (*ii*) *spatial Gillespie* simulators represent space with a coarse mesh and represent chemical concentrations with integer-valued populations within each sub-compartment, and (*iii*) *particle-based* simulators represent space continuously and represent molecules as point-like particles. Of these, particle-based simulators are used most widely [5] because they offer the best spatial resolution, they do not introduce artifacts from an underlying spatial lattice, and biologically realistic membrane geometries can be modelled easily.

MCell [15] and Smoldyn [2] are the most popular particle-based simulators. MCell has been used extensively to investigate neural signaling; for example, Coggan and coworkers used MCell to show that much neural signal transmission does not occur at synapses but at relatively distant sites [7]. Smoldyn has been used primarily to investigate signal transmission and processing with *Escherichia coli* cells. For example, Lipkow and Odde used Smoldyn to show that a combination of spatially localized protein modification and modification-dependent protein complexation can produce intracellular concentration gradients [11]. The primary downside to both of these particle-based simulators is that they are very computationally intensive.

Smoldyn runs substantially faster than MCell for two reasons. First, Smoldyn achieves about a factor of 2.5 better speed through more efficient algorithms [2]. For example, Smoldyn simultaneously diffuses all molecules with random displacements and then executes bimolecular chemical reactions for all pairs of reactants that end up within a precomputed *binding radius* [4] of each other. In contrast, MCell diffuses one molecule at a time with a random displacement and executes a bimolecular reaction if any point along the straight-line trajectory of this displacement intersects another molecule [10]. Secondly, Smoldyn's algorithms produce more accurate results than do MCell's, for the same size simulation time step [2]. Thus, for a given level of accuracy, Smoldyn can use longer time steps and hence run faster.

Smoldyn is a particle-based spatio-temporal chemical reaction dynamics simulator. Chemical molecules reside in a well defined domain and diffuse by Brownian motion. Diffusion can be isotropic, anisotropic or drift (diffusion under the influence of external fields). There are three types of chemical reactions: zero-order reactions ($\phi \rightarrow S_a$) handle molecule additions to the system from some un-modelled process (example: infinite source), uni-molecular reactions ($S_a \rightarrow \phi, S_a \rightarrow S_b, S_a \rightarrow S_b + S_c,$) simulate disassociation and conversions, and bi-molecular reactions ($S_a + S_b \rightarrow S_c + S_d, S_a + S_b \rightarrow S_c, S_a + S_b \rightarrow \phi$) simulate association, mutual destruction and other bimolecular chemical reactions. It also simulates interaction between molecules and geometry such as absorption, reflection, and transmission. Reactions can be restricted to occur on a surface alone. Various types of implicit and explicit (mesh) surfaces are supported. The simulation advances with a pre-determined time step δt.

In this paper we present results of our *preliminary work* to address the computational complexity. We have developed data-parallel algorithms for enabling execution of Smoldyn on graphics processing units (GPUs). In our preliminary work, we restrict our implementation to simulation inside a cubic domain without geometry. We implemented methods for diffusion, zero-order, uni-molecular, and bi-molecular reactions. Our results indicate a substantial speed-up over the serial implementation without loss of accuracy. The performance gain will enable efficient simulation of realistically sized models. The rest of the paper is as follows. In section 1.1 we briefly introduce GPU-based parallel computing. In section 2 we describe our novel algorithms and data structures. In the results section, we present results of our accuracy test and benchmarks. In section 4 we discuss our results and future work.

1.1. Scientific Computing on Graphics Processing Units(GPUs)

Graphics Processing Units (GPUs) were primarily developed to accelerate rendering computations in computer graphics. Initially, the so-called *graphics pipeline* was of fixed functionality and tightly bound to the hardware. The need for specialized user-defined rendering schemes led GPU vendors to enable programmability. Computational scientists used this programmability for accelerating certain scientific computations [13]. However, they had to reformulate these computations in terms of graphics rendering. Since 2007, GPU vendors such as NVIDIA have developed direct API to facilitate high performance computing on GPUs. Currently, there are many APIs such as NVIDIAs CUDA, OpenCL, and Microsoft's Direct Compute. Simultaneously, GPU hardware has also evolved from its early graphics specific nature to handle more general data-parallel computing tasks. The latest GPU from NVIDIA has a peak performance of 1.5 T Flops (single precision), supports IEEE574-2008 double precision arithmetic, and has error correction (ECC). It has a memory bandwidth of nearly 144GB/s. Moreover, the inclusion of L1, L2 cache in addition to shared memory (user-defined cache) means that certain scientific algorithms with irregular data-access patterns can be handled much more efficiently. This GPU has a 25x computing power advantage and a 2x memory bandwidth advantage over the latest generation quad-core CPUs. However, due to the programming model, unlike traditional parallel programming specifications such as OpenMP, code written for serial execution cannot be incrementally parallelized on GPUs. Parallel execution of GPUs requires fundamentally different algorithms and data structures to take advantage of the architecture. This is the main challenge that we address in this paper.

2. SMOLDYN ALGORITHM

The basis of the Smoldyn method is shown in Algorithm 1 At the beginning of simulation, the state of the molecules $\{p_i\}$ is initialized. While initial positions p_i are chosen such that there is a uniform distribution of particles in the computational domain, the particles are then diffused through a time step δt according to their diffusion coefficients. Then, collisions with boundaries are evaluated. We use specular, periodic and absorb boundary conditions in this implementation. The next steps are processing of zero-order and uni-molecular reactions. The molecules are then partitioned into their respective domain cells and indexed by cell. This step is necessary for the processing of the bi-molecular reactions. At each step we process particles concurrently. Finally, we execute various user commands and sample statistics, if necessary.

Algorithm 1 Smoldyn Algorithm

```
1: initialize state of the system $S = \{p_1^0, p_2^0, .........p_N^0\}$
2: while $t < t_f$ do
3:     process molecular diffusion
4:     resolve collisions with boundaries
5:     process zero-order reactions
6:     process uni-molecular reactions
7:     index particles into cells
8:     process bi-molecular reactions
9:     execute user commands
10:    sample properties
11: end while
12: process and output sampled data
```

3. DATA-PARALLEL ALGORITHMS FOR GPU-BASED EXECUTION

GPUs have a few important architecture specific limitations. First, memory read/write operations from global memory have to be coalesced to prevent serialization. At the software level, threads are organized into thread blocks. Each thread block is executed on a single multi-processor. There are 16 multi-processors in a NVIDIA Fermi GPU. There is a limitation on the maximum number of threads in a thread block as well as the maximum number of thread blocks that a multi-processor can handle. At the hardware-level, threads are organized into warps, each containing 32 threads. All threads in a warp have a single program counter and execute in lock-step (single instruction multiple data (SIMD)). This means that program-path divergence between threads in a warp should be avoided. Finally, dynamic containers such as lists, where elements can be added one at a time, are extremely inefficient on GPUs because it is not possible to allocate memory one unit at a time. Our algorithms are designed to work around these limitations. In the following sections, we describe our algorithms and associated data-structures.

3.1. Data Structures

We store molecule state (position and type) in an array of structures (float3 and int). When the *type* field is set to -1, it indicates a dead molecule or free space. This structure takes 16 bytes in storage, so memory access is coalesced. We calculate molecular moments (first and second order) as a part of statistics gathering and we use a structure of 2 float3 values to store moments for each molecule (See Fig. 1, where N is a total number of molecules). In addition, we use two additional arrays of unsigned integers to sort the molecule index and cellID. Molecule index is the location in the data array where the state of the molecule is stored. The simulation domain is divided into cells. CellID is essentially the 'name' of the cell where a molecule is physically located. The cell data is stored in a separate array consisting of two integers, cellStartIndex and cellEndIndex (see Fig. 2, where R is a number of cells). These integers are start and end location for each cell in the sorted molecule state array. Naturally, the number of molecules in a cell is given by $N_c = \text{cellEndIndex} - \text{cellStartIndex}$.

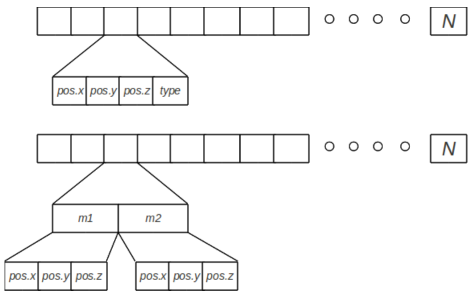

Figure 1. Molecule Data (top - molecule state, bottom - molecule momentum)

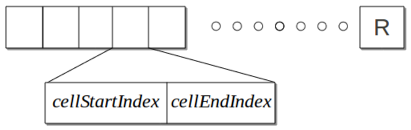

Figure 2. Cell Data (start and end indices for each cell)

3.2. Random Number Generator

One of the most important components of any reaction-diffusion implementation is the psuedo-random number generator (pRNG). It is desirable for the pRNG to have a large period, a small state, and fast execution. The Mersenne Twister pRNG (MTpRNG) designed by Matsumoto and Nishimura [12] has extremely good statistical quality. Internally, it uses a binary matrix to transform one vector of bits into a new vector of bits, using an extremely large sparse matrix and large vectors [9]. Its period is 2^{19937}; however, it requires 84 bytes to store the state, and the state should be updated serially and can not be parallelized. This pRNG requires too much memory for the state, which is not suitable for execution on GPU due to memory limitation (84 bytes cannot be fit into registers and local memory is highly inefficient). Because of this limit, we use another generator in our simulation. Inside the simulation loop we use the Xor 128 random number generator [6]. This generator has a period of $2^{128} - 1$, and requires only 16 bytes (4 32-bit integers) to store its state and can be easily fit into register memory. We use a combination of four streams of xorshift generators, each of which has a period of $2^{32} - 1$. Each independent stream of xorshift generator is generated using several bitwise instructions. Such an approach allows us to hide any statistical defects in each generator and obtain a period of $2^{128} - 1$. For GPU execution, it is not feasible to run a single pRNG. Each thread of execution must have its own independent pRNG. To prevent the emission of correlated sequences by each generator, we generate different initial states for each thread using a high quality random number generator on CPU. This approach allows us to achieve good randomness across different threads.

3.3. State Initialization

The simulation is initialized by instantiating molecules within the simulation domain based on parameters in a configuration file. This distribution can be random or uniform in any of the three possible directions. For each type of particle, we launch a separate kernel to instantiate the molecules. In the Fermi architecture several kernels can be launched in parallel. Since the initial number of particles is known for each type, the exact region in the molecule array where each kernel operates is clearly known (See Fig. 3, where M is a number of blocks). We also maintain a single pointer to the location of the last non-empty location in the molecule array. We have each thread in a block process one particle. Typically, we have 128 threads in a block, and as many blocks as we need to generate the desired number of particles in one kernel invocation. Since we can have different distribution type in each direction, we use template meta-programming to resolve this. We encapsulate code that distribute molecules in a particular direction in a functor and pass it as a kernel parameter depending on the configuration file.

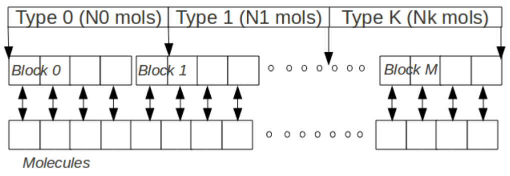

Figure 3. Molecule generation

3.4. Molecular Diffusion

The molecular diffusion step is performed at every time step. Molecules diffuse according to their diffusion coefficients. Diffusion constants are stored in constant memory and are indexed by molecule type. Constant memory is cached. This process significantly reduces overhead when multiple threads working on different molecules of the same type access the same value. We then use gaussian random displacement for each molecule to get the direction and magnitude of movement.

3.5. Zero-order reactions

Zero-order reactions create particles at every time step based on reaction parameters. We pre-calculate the number of molecules that need to be generated for each reaction. We then launch independent kernels for each zero-order reaction. During this step we do not physically reallocate memory, but just convert dead (unused) particles into particles of desired type. Just as in the initialization stage, every kernel is aware of the exact location where these molecules will be instantiated in the molecule array. Each kernel updates the memory pointer to the last live molecule in the array (Fig. 4).

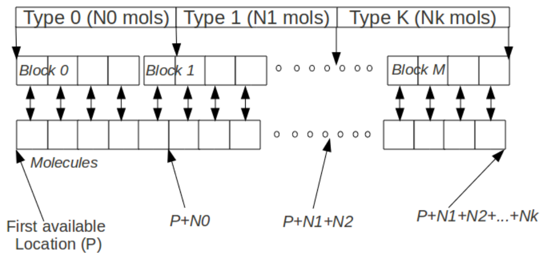

Figure 4. Zero-order reactions

3.6. Uni-molecular Reactions

Uni-molecular reactions have three types: disintegration ($S_a \rightarrow \phi$), mutation ($S_a \rightarrow S_b$), and disassociation ($S_a \rightarrow S_b + S_c$). In terms of memory management, mutation is straightforward to handle. We just change the type and position of the current molecule object in the molecule state array (see Fig. 5) without any memory reallocation. Based on the previous location, we find the cell in which the current molecule resides. The mutated molecule's position is set to a random location within the same cell.

Disintegration involves memory deallocation. A simple way to handle this situation would be to update the molecule type to indicate a dead molecule. However, this method will create empty slots in the molecule data array at random locations as the simulation progresses, which will fragment the molecule state array. One way around this challenge is to sort the molecule state array based on type and then collect all empty spots at one end. This process has to be done at every time step and can be very expensive. Instead of sorting, we do stream compaction, using Thrust algorithm Remove If, which moves dead particles to the end of the array. This algorithm is very well optimized and does not add any significant overhead.

Disassociation involves a molecule mutation and creation of a new molecule simultaneously. Molecule mutation is handled as in the case of pure mutation. For molecule creation we maintain a pointer to the location where the empty locations start. When disassociation occurs, we simply add a molecule after the last one. We use an atomic increment operation on the pointer, with each thread of execution maintaining a local copy of the pointer to accomplish this consistently. New molecules are thus added to the end of the molecule state array (see Fig. 6).

As is the case with zero-order reactions, each reaction is handled with a separate kernel launch. While all possible uni-molecular reactions could be potentially handled by a single kernel, the stochastic nature of the algorithm combined with the requirement for each thread of execution to search a large reaction table to find reaction rates and products can cause significant branch divergence and therefore loss of performance. On the other hand, kernel launch overhead is minimal on the latest GPUs. Furthermore, multiple independent kernels can be launched in parallel, increasing the parallelism.

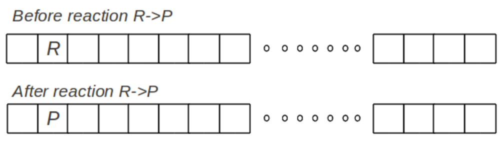

Figure 5. Uni-molecular reactions. Single product

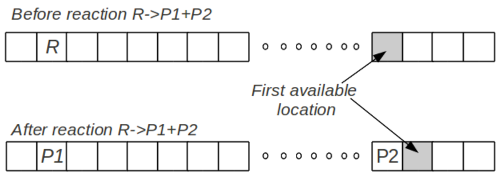

Figure 6. Uni-molecular reactions. Double products

3.7. Bi-molecular Reaction

Bi-molecular reactions, as the name suggests, involve two reacting molecules. These two molecules have to be within a certain *binding radius* in space to be able to react. Therefore, in addition to the reaction specifications, there are also space considerations. The spatial hashing scheme we use is described in CUDA "particles" sample [8]. We will briefly cover the details for completeness. The overall process is divided into three kernels. In the first kernel each thread computes the cellID according to equation 1. Here i, j, k are cell ID in X,Y and Z direction respectively. Each thread in a block processes one or more molecules in a coalesced manner. The launch configuration we use is 256 thread blocks with 128 threads in each. See Fig. 7 for more details. The second kernel uses a radix sort to sort the array based on cellID, in order to group all particleID's with the same cellID together. The speed of the radix sort [14] is dependent on the number of bits specified. The algorithm is illustrated on Fig. 8, where C is a number of cells - 1. The final step is to find the start (cellStartIndex) and end (cellEndIndex) location in the molecule array for each cell. Each thread loads the cellID associated with the current particle in shared memory and compares it with the previous molecule cellID. If both of these hashes differ, then it indicates the beginning of a new cell (see Fig. 9, where C is a number of cells - 1). In this algorithm we use shared memory to achieve memory coalescing.

$$\begin{aligned} i &= (int)\frac{pos.x - \text{gridMin.x}}{\text{gridWidth}} \\ j &= (int)\frac{pos.y - \text{gridMin.y}}{\text{gridHeight}} \\ k &= (int)\frac{pos.z - \text{gridMin.z}}{\text{gridDepth}} \end{aligned} \tag{1}$$

$$\text{CellID} = k * \text{gridDim.x*gridMin.y} + j * \text{gridMin.x} + i \tag{2}$$

The size of the cells is a little bit larger than a reactant molecule's *binding radius*. Therefore, for a given molecule, a search has be conducted for potential reacting molecule(s) in the nearest 27 neighbouring cells. If there is more than one molecule, the closest molecule is chosen. This scheme is the high accuracy scheme described in the original Smoldyn implementation [2]. In our current implementation, we use the low accuracy scheme where the search of the second reactant molecule is restricted to the same cell as the first molecule. This scheme is a part of original Smoldyn as well. We use one thread per cell when running this kernel, which can lead to clashes in the high accuracy scheme between threads that select the same reacting molecules. For example, 2 kernels could process the same molecule while processing neighboring cells, which would led to inaccuracy. In the low accuracy scheme this clash is avoided.

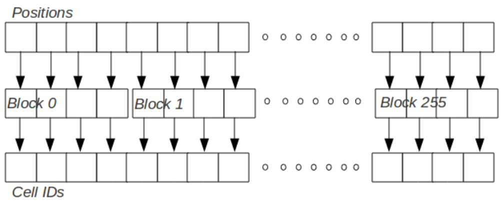

Figure 7. Cell IDs generation

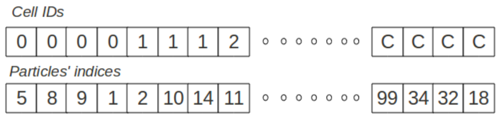

Figure 8. Radix sort

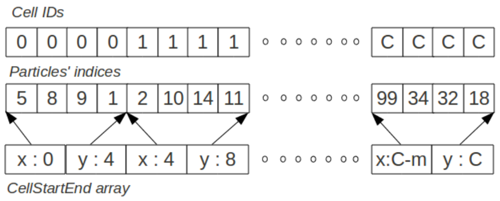

Figure 9. Kernel to find cell start and end indices

Bi-molecular reactions can be of three types: mutation $S_a + S_c \rightarrow S_d + S_c$, association $S_a + S_b \rightarrow S_c$, and disintegration $S_a + S_b \rightarrow \phi$. Mutation is handled as in the case of the uni-molecular reactions. Association involves killing off one molecule and mutating the other. Disintegration involves destruction of two molecules. All these are handled in memory using *atomic* operations.

3.8. Resolving Boundary Conditions

We have a separate kernel for resolving boundary conditions. The kernel reads molecular position from global memory and processes boundary conditions according to the configuration file. We use function pointers in these kernels to handle different boundary conditions (periodic, reflective, transparent, and absorptive) which eliminates branching within the kernels. Each molecule is checked for boundary interaction and appropriately processed.

3.9. Statistics Sampling

Sampling statistics involves finding quantities such as the number of molecules of a certain type or the center of mass of all molecules of a certain type and the first moment. We use a library developed by NVIDIA called Thrust to implement statistics. At its core, Thrust uses parallel pre-fix sum to accomplish reduction in parallel of large vector sets. Thrust is a C++ compatible library that one can use to specialize the binary operations in the parallel pre-fix sum algorithm.

3.10. Visualization

As we have OpenGL-based visualization in our simulation, we use Vertex Buffer Objects (VBO) to store molecules' positions and colors. These objects can be used by CUDA as well as by OpenGL and provide very efficient drawing operations.

4. RESULTS

We benchmarked our implementation against the original Smoldyn implementation. In our benchmarks the serial implementation was executed on an Intel iCore 7 processor and our parallel implementation was executed on an NVIDIA Fermi 2050 Tesla. We used boundary conditions of one of the following types: Absorptive (a particle disappears if it encounters the boundary), Transparent (a particle can pass through the boundary), Periodic, and Reflective. These conditions are defined in a configuration file. We expected 4-5 particles per cell, so we subdivided the simulation space accordingly. We defined the number of cells in each direction as: $\sqrt[3]{\frac{N_{mols}}{4}}$.

For performance benchmarks, we simulated molecule counts ranging from 30000 to 3000000 (see Fig. 14). We ran different models to test performance advantage of our implementation for different cases. These systems included diffusion, zero-order, uni-molecular, and bi-molecular reactions. In these benchmarks we tested overall Smoldyn algorithm execution time, not just isolated algorithms, running the same configuration files with CPU and GPU versions.

We also conducted statistical benchmarks comparing output of our implementation with theoretical expected values. We conducted the following experiments:

Diffusion benchmark

For the diffusion test we used a system with molecules of 3 types with different diffusion coefficients. Fig. 10 compares the output of our system with the theoretical graph. Simulation outputs are denoted with symbols and theoretical results for the same parameters are denoted with solid black lines. Mean square displacements of three populations of freely diffusing molecules are presented.

Zero-order reactions benchmark

For the zero-order reactions test, we used a system with 3 zero-order reactions, generating molecules of 3 types. The reactions we tested are:

$$\phi \rightarrow red$$
$$\phi \rightarrow green$$
$$\phi \rightarrow blue$$

We ran the system for 8 simulation seconds with a time-step of 0.1 sec. Variations in the number of molecules over time are presented in Fig. 11. Theoretical results are represented with solid lines.

Uni-molecular reactions benchmark

For the uni-molecular reactions test, we used a system with molecules of 3 types and with 3 uni-molecular reactions. The reactions we tested are:

$$red \rightarrow \phi$$
$$green \rightarrow \phi$$
$$blue \rightarrow \phi$$

This system ran for 3 simulation seconds with a time-step of 0.1 sec. The initial number of molecules was 1000 of each type. Graph 12 shows variations in the numbers of molecules over time. The solid lines are theoretical expectations.

Bi-molecular reactions benchmark

For the bi-molecular reactions test, we used a system where two molecules, each of a different type, consume each other. The reaction we used is:

$$AH + B \rightarrow \phi$$

The system contained 2468 molecules of type AH and 120468 molecules of type B. The graph 13 shows variations in the numbers of molecules AH over time. We ran this system for 0.1 seconds with a time-step of 0.002 seconds.

5. DISCUSSION

As can be seen from the benchmark results (Fig. 14), our results show that performance grows as the model size grows. There is a break-even point (not shown), where the GPU implementation is becomes faster than the CPU implementation. At smaller model sizes, due to lower parallelism, a significant portion of the GPU is not utilized. According to our experiments, speed-up reaches 100 for systems with approximately 1 million molecules. The highest speed-up was achieved for a simple model with just a molecular diffusion and the lowest was achieved for systems with bi-molecular reactions. These results correspond to the complexity of these algorithms. Also, all reactions benchmarks involved diffusion and therefore the speed-up a model with just a diffusion cannot be higher than a speed-up for a system with molecular reactions.

The tests we used for statistical benchmarks were the same tests that original Smoldyn authors used to validate the non-parallel version of Smoldyn [2]. All of these simulation results agreed well with analytical theory. This shows that the individual algorithms of this GPU-accelerated version of Smoldyn are quite accurate both at and away from steady state. As with non-parallel Smoldyn, simulations that combine multiple algorithms yield results that are necessarily approximate, although they approach exactness as time steps are reduced towards zero. This proves that our model performs correctly.

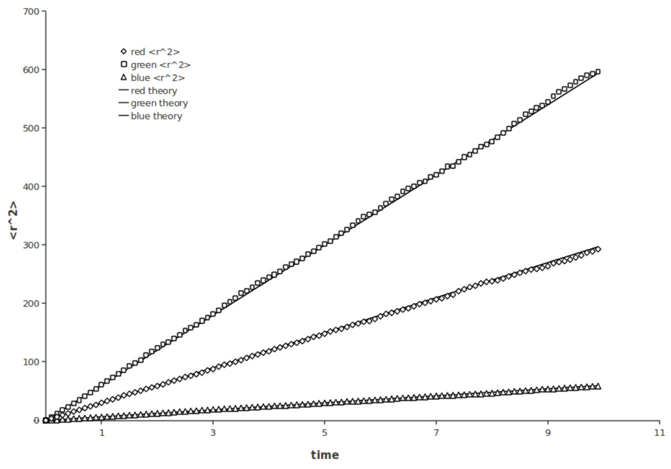

Figure 10. Diffusion statistical benchmark

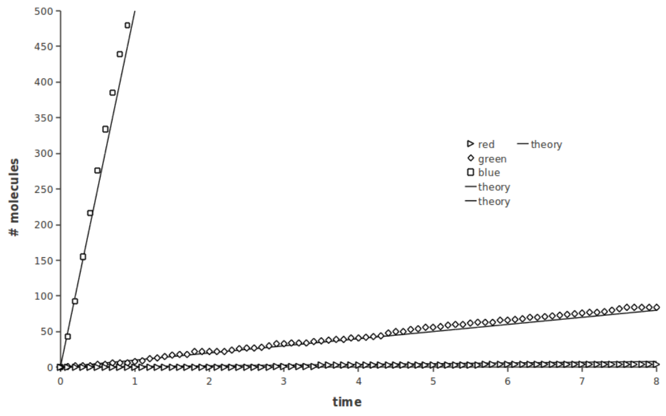

Figure 11. Zero-order reactions statistical benchmark

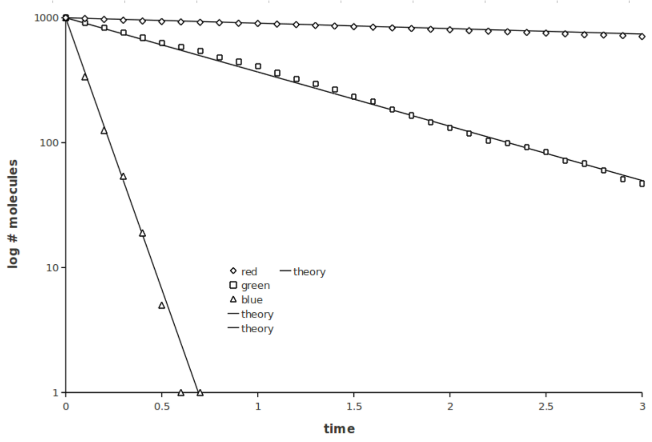

Figure 12. Uni-molecular reactions statistical benchmark

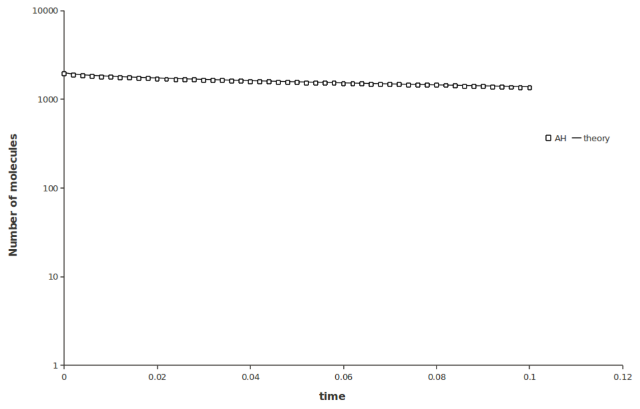

Figure 13. Bi-molecular reactions statistical benchmark

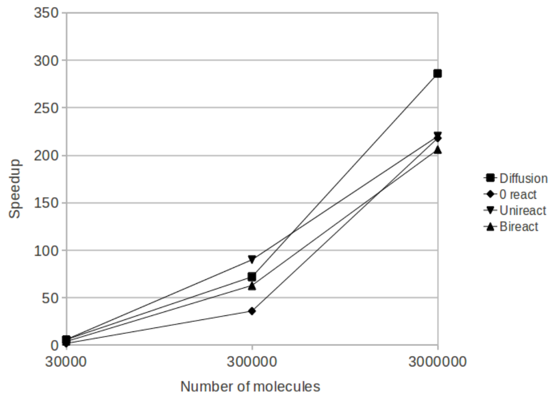

Figure 14. Performance benchmark

6. CONCLUSIONS AND FUTURE WORK

We have successfully implemented a GPU-accelerated version of the Smoldyn solver for spatio-temporal chemical kinetics. Currently, our solver is capable of handling zero-order, uni-molecular, and bi-molecular reactions. All boundary conditions that are specified in the original Smoldyn implementation can be handled. Our benchmarks against the original serial implementation show a performance gain of 200x. This will enable faster simulation of much larger systems than would have previously been possible. Our implementation is capable of handling systems with up to 16 million molecules on a single GPU. Larger systems may have to go to a mulit-GPU execution, which we will investigate in the near future. Currently, we are working towards incorporating complex cellular geometry within the simulation space. We plan to use a separate spatial geometry hash for this purpose. We will also implement molecule diffusion and reactions on surfaces. This approach could potentially be tricky for complex mesh surfaces as molecule trajectories could pass through vertices. Simultaneously, we are developing an on-line service where users can submit configuration files and obtain results through email. In the near future, we plan to implement this service on GPU-clouds.

7. ACKNOWLEDGMENTS

This work is funded through National Science Foundation (NSF) grants CCF - 1013278 and CNS - 0968519. Any opinions and conclusions listed are those of the authors' alone and do not necessarily represent the view of the NSF. The authors also would like to thank Dr. Marjorie Piechowski for help with preparation of this paper.

REFERENCES

[1] R. Alves, F. Antunes, and A. Salvador. Tools for kinetic modeling of biochemical networks. *Nat. Biotechnol.*, 24:667–672, 2006.

[2] S.S. Andrews, N.J. Addy, R. Brent, and A.P. Arkin. Detailed simulations of cell biology with smoldyn 2.1. *PLoS Comput. Biol.*, 6:e1000705, 2010.

[3] S.S. Andrews and A.P. Arkin. Simulating cell biology. *Curr. Biol.*, 16:R523–R527, 2006.

[4] S.S. Andrews and D. Bray. Stochastic simulation of chemical reactions with spatial resolution and single molecule detail. *Phys. Biol.*, 1:137–151, 2004.

[5] S.S. Andrews, T. Dinh, and A.P. Arkin. Stochastic models of biological processes. In R.A. Meyers, editor, *Encyclopedia of Complexity and System Science*, volume 9, pages 8730–8749. Springer, New York, 2009.

[6] R. P. Brent. Note on marsaglia's xorshift random number generators. *J. Statis. Softw*, 11:1–4, 2004.

[7] J.S. Coggan, T.M. Bartol, E. Esquenazi, J.R. Stiles, S. Lamont, M.E. Martone, D.K. Berg, M.H. Ellisman, and T.J. Sejnowski. Evidence for ectopic neurotransmission at a neuronal synapse. *Science*, 309:446–451, 2005.

[8] S. Green. Cuda particles. White paper, NVIDIA Corp, 2008.

[9] L. Howes and D. Thomas. Efficient random number generation and application using cuda. In *GPU Gems 3*, pages 805–829, 2008.

[10] R.A. Kerr, T.M. Bartol, B. Kaminsky, M. Dittrich, Jen-Chien J. Chang, S.B. Baden, T.J. Sejnowski, and J.R. Stiles. Fast monte carlo simulation methods for biological reaction-diffusion systems in solution and on surfaces. *SIAM J. Sci. Comput.*, 30:3126–3149, 2008.

[11] K. Lipkow and D.J. Odde. Model for protein concentration gradients in the cytoplasm. *Cellular and Molecular Bioengineering*, 1:84–92, 2008.

[12] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[13] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, and A.E. Lefohn. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[14] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for many core gpus. In *Proc. of 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[15] J.R. Stiles and T.M. Bartol. Monte carlo methods for simulating realistic synaptic microphysiology using mcell. In E. De Schutter, editor, *Computational Neuroscience: Realistic Modeling for Experimentalists*, pages 87–130. CRC Press, Boca Raton, FL, 2001.

[16] K. Takahashi, S.N.V. Arjunan, and M. Tomita. Space in systems biology of signaling pathways towards intracellular molecular crowding in silico. *FEBS Lett.*, 579:1783–1788, 2005.