Improving reaction kernel performance in Lattice Microbes: particle-wise propensities and run-time generated code

Michael J. Hallock*, Zaida Luthey-Schulten†‡§
*School of Chemical Sciences, University of Illinois at Urbana-Champaign, Urbana, Illinois
†Department of Chemistry, University of Illinois at Urbana-Champaign, Urbana, Illinois
‡Department of Physics, University of Illinois at Urbana-Champaign, Urbana, Illinois
§Beckman Institute, University of Illinois at Urbana-Champaign, Urbana, Illinois

***Abstract*—The reaction kernel for MPD-RDME, the GPU-accelerated reaction-diffusion master equation solver found in Lattice Microbes uses a large number of kinetic parameters to describe a biochemical network. Many of these parameters are required to compute the system's total reaction propensity, which is used to stochastically evaluate whether a reaction event takes place. In this paper, we examine two techniques for accelerating performance by modifying the total propensity calculation. The first technique is to use a particle-based approach to compute propensities from discrete particles and particle pairs. We find this technique results in a dramatic improvement in performance for a complex model, approximately 60 times faster. The second technique uses run-time generated source code to automatically create executable code tailored for the biological model being simulated. The removal of all memory reads for constant parameters increases performance for less complex models.**

I. Introduction

Reaction-diffusion processes are common in biology, and simulation of these processes is a computationally expensive task. As simulations of biological processes in spatially-inhomogeneous (such as the crowded cellular cytoplasm) environments become more widely important to the field of Systems Biology, the need arises for high-performance solvers for the reaction-diffusion master equation. A wide array of tools are currently available to researchers to perform this task: Smoldyn [1], Mcell [2], ReaDDY [3], and MesoRD [4], to name a few. MPD-RDME, the GPU-accelerated reaction-diffusion master equation solver found in Lattice Microbes [5] offers the ability to perform simulations of biologically relevant size and time scales, using one or more GPUs [6]. This algorithm is the focus of this work, and modifications to the reaction kernel to increase performance will be explored.

A model of a biochemical process requires many parameters to describe all the chemical species present in a simulation and the set of reaction rules that governs the interaction between species. Each reaction defines a set of species that a reaction requires (reactants) to be present in order to occur and a set of species that represents the products of the reaction. These reactions can come in one of three elementary types that follow a specific chemical rate law. A *first order* reaction is one that depends on the concentration of one type of chemical species, a *second order* reaction that depends on the concentration of two species, and a *zeroth order* that depends only on time. Each reaction has a single rate parameter that controls how frequently that reaction can occur. Given a reaction and the quantity of the chemical species present, one can compute the propensity function (a_r) that represents the probability of that reaction occurring over some interval of time Δt.

In the MPD-RDME algorithm, we treat our simulation volume as a regular rectilinear lattice of voxels (referred to as a lattice site) each containing a finite number of particles, representing a chemical species. The algorithm uses operator splitting to calculate the reaction and diffusion events in the simulation separately. The algorithm combines the multiparticle (MP) method for diffusion developed by Chopard et al. [7] with Gillespie's stochastic simulator algorithm [8] for reactions within lattice sites. This approach is similar to the Gillespie multi-particle (GMP) method first introduced by Rodrguez et al. [9]

The lattice site is considered to be well-stirred such that we do not track the exact position of a particle and consider that there is equal chance to interact with any other particles in the same site. Particles probabilistically diffuse into neighboring sites based on diffusion rate parameters, and is performed via the diffusion operator [10]. Each site is assigned a "site type" that represents something about the physical environment that site is representing. Typical site types are extracellular space, cytoplasm, membrane, nucleoid, etc. Diffusion rates between different site types can be defined, and reactions are assigned to occur only certain site types. This allows one to constrain reactions to only occur in a certain environment.

The reaction operator, shown in Algorithm 1 for a single lattice site, can be divided into four sections. The first part is accumulating the total propensity (a_{tot}) for a reaction to occur, the second is to determine if a reaction will occur during this timestep given the propensity, third to determine which reaction will occur, if more than one reaction could possibly occur, and lastly to update the particles in the site to reflect the reaction event occurring.

MPD-RDME supports zero, one, or two reactants producing any number of products. Each reaction is defined via a number of parameters that are fixed throughout the lifetime of the simulation. Each reaction r in the set of reactions R define the reactants (r_{r1} and r_{r2}), the products ($r_{p1...pN}$), the propensity constant derived from the rate, representing the probability of this reaction occurring during a timestep (r_{prop}), the site types the reaction occurs in (r_ν), and the reaction order (r_{order}) that represents the rate law used to describe this reaction.

In memory, we store four integer vectors of length $N_{\text{reactions}}$ to store the first reactant, second reactant, the reaction order, and the reaction propensity for each reaction. The reaction location (RL) is a $N_{\text{reactions}} \times N_{\text{sitetypes}}$ matrix of 1-byte values to indicate where reactions can take place. The stoichiometric matrix (S) is a $N_{\text{reactions}} \times N_{\text{species}}$ matrix. It contains a row for every reaction and the value in the columns are the change in quantity of the corresponding chemical species as a result of that reaction. Species that are consumed in the reaction are negative values, and those that are produced are positive values. All other species have a zero value indicating no change in their respective counts occur. As all these parameters are fixed for the lifetime of a simulation, we store these four vectors and the S and RL matrices in GPU constant memory.

GPU constant memory is advantageous because all threads in a half-warp can simultaneously receive a value from constant memory if all threads are accessing the same element at the same time. Work on the GPU is divided such that each thread is processing a single lattice site. All threads in the warp compute the propensity for one reaction at the same time, thus all accesses to constant memory across the entire warp are to the same element and will occur without serialization. This makes the algorithm very amenable for GPU computing due to predictable memory accesses across a warp and there is minimal divergence, until the decision if a reaction occurs for a lattice site is reached.

```
for $r \in R$ do
    $a_{tot} = a_{tot} + a_r(\mathbf{x}_\nu)$
end
$n_1 = uniformRand()$
if $n_1 \leq \int_0^\tau a_{tot} e^{-a_{tot} t} dt$ then
    $n_2 = uniformRand()$
    for $r \in R$ do
        if $a_{r-1}(\mathbf{x}_\nu) < n_2 \cdot a_{tot} \leq a_r(\mathbf{x}_\nu)$ then
            perform reaction $r$ in subvolume $\mathbf{x}_\nu$
        end
    end
end
```

Algorithm 1: The reaction operator in MPD-RDME[5]. Shown for a single lattice site ν in the lattice.

GPU constant memory is limited to a fixed size of 64 kB

```
if $siteType_\nu != r_\nu$ then
    return 0
end
$count1 = 0$
$count2 = 0$
for $p \in (\mathbf{x}_\nu)$ do
    if $p = r_{r1}$ then
        increment $count1$
    end
    if $p = r_{r2}$ then
        increment $count2$
    end
end
if $r_{order} = Zeroth$ then
    return $r_{prop}$
else if $r_{order} = First$ then
    return $r_{prop} * count1$
else if $r_{order} = Second$ then
    return $r_{prop} * count1 * count2$
end
```

Algorithm 2: Procedure a_r for computing the propensity of a given reaction r

of data. The current number of different chemical species that can be defined for an MPD-RDME simulation is 255. If one wishes to use the maximum number of species, then the size of the S matrix is 255 times the number of reactions. Sixteen bytes are needed for each reaction across the reactant, order, and propensity vectors. For RL, one byte per reaction is needed per site type. Assuming the minimum of a single site type, each reaction requires 272 bytes. This could allow up to 290 different reactions to be defined for a simulation if constant memory was devoted to reactions, however in practice we also store diffusion-related parameters in constant memory and typically restrict S and RL to 16 kB and 10 kB, respectively.

Our simulation of ribosome biogenesis [11] involved 884 reactions. In order to overcome the size restriction of GPU constant memory, the S and RL matrices were moved to GPU global memory. The remaining four vectors (first reactant, second reactant, order, propensity) remained in constant memory. This scheme could support up to $4,096$ reactions as sixteen bytes of GPU constant memory are required per reaction. To go beyond that, we could continue to move all parameters into GPU global memory, where then the total number of reactions will be limited only by available free memory.

Every thread processing a lattice site must compute the total propensity (a_{tot}) as the first step of algorithm 1. The procedure for this is shown in algorithm 2. Typically, very few lattice sites have a reaction occurring in them during a timestep. There may be no reactions defined for a given site type, or the lattice site is empty or contains only non-reacting particles. Therefore, computing the propensity quickly is performance critical, and the placement of parameters in the GPU memory hierarchy may have performance side-effects. In this paper, we will evaluate two alternate approaches to computing the propensity for a lattice site. The first approach is to reformulate the algorithm to sum propensities based present chemical species in a lattice site, rather than on a reaction-by-reaction basis, which trades a larger number of uniform memory accesses for a smaller number of non-uniform loads. The second approach is to use use just-in-time compilation as a means of turning the constant parameters into immediate loads, embedding them directly into code to be compiled at run-time.

A. *Test Models*

Three representative models are used to test the different reaction kernels. These models represent a range of complexities. The first model is a simple bimolecular reaction model, to serve as a test of a model with little complexity. The second is the *E. coli* lac genetic switch, as described in the Lattice Microbes Instruction Guide [12]. The third is a model of *E. coli* ribosomal biogenesis [11], which is the most complex model. The relevant statistics of each model can be found in Table I.

The bimolecular model is a $2\,\mu m \times 2\,\mu m \times 2\,\mu m$ box that is one homogeneous site type. The Lac genetic switch model is a full-sized typical *E. coli* cell of $1\,\mu m \times 1\,\mu m \times 2\,\mu m$ with a membrane, cytoplasm, and extracellular site types. The ribosome biogenesis model is $1\,\mu m \times 1\,\mu m \times 4\,\mu m$, representing the doubled-cell-size present in an *E. coli* undergoing cell division. It contains a nucleoid region, cytoplasm, membrane, and extracellular site types. All models use a periodic boundary condition at the lattice edges.

The performance results for these three models will be measured by comparing the average wall-time of the original reaction kernel to the wall-time of the alternate kernels described below. The GPU used for these tests is a single NVIDIA TITAN-X, a high-end consumer-grade GPU.

B. *S and RL in GPU Global Memory*

In order to simulate models with a large number of reactions, we first needed to overcome the small amount of GPU constant memory made available to applications. We achieve this by relocating the stoichiometric (S) and reaction-location (RL) matrices to GPU global memory. The change is straightforward; memory is dynamically allocated for these data structures in GPU global memory and the their addresses are passed as arguments to the kernel, as opposed to accessing them via GPU constant memory. On hardware that supports it, the read-only data cache load intrinsic `__ldg` [13] is used to fetch values from these matrices. This alternative way of accessing memory is helpful for non-coalesced reads as the texture cache is used, without the complexity of actually binding these data structures as textures.

This version of the code will be used for comparisons of performance, otherwise there is no basis for comparison for the ribosome biogenesis model. For the Bimolecular system, there is no difference in performance regarding the placement of the S and RL matrices. For the Lac switch, there is a degradation of performance of roughly 7%, changing the average wall-time for the reaction kernel to run from .246 ms to .264 ms.

II. Particle-wise Propensity Summation

The first approach to accelerating the kernel is to consider a different algorithm for computing a lattice site's total reaction propensity a_{tot} entirely. Instead of computing the total propensity as a sum of all the individual reaction propensities, we can consider instead summing the propensity of all possible interactions of the particles in a given lattice site. For example, consider a model with a first-order reaction for A to B with propensity of occurring in a timestep equal to p_1, and a second order reaction for A and C forming D with a propensity of p_2. If in a lattice site, we had three A particles and two C particles, the total propensity is the sum of the two reaction's propensities according to their rate laws: $3 * p_1 + 3 * 2 * p_2$. Alternatively, we can consider individually the propensity of each particle in a lattice site and all combinations of two particles and sum those. Each A particle contributes p_1, since there is a first-order reaction defined for A; each C individually contributes nothing. For second-order reactions, all particle pairs are examined. Combinatorially there are six AC pairs that each have a propensity p_2; the AA and CC pairs have zero reaction propensity. In total, it is the same propensity as it is when computed on a reaction-by-reaction basis.

To compute the total propensity for a lattice site, we will look up values from three pre-computed data structures: A vector for the zeroth-order propensities for each site type (A^0), an array for the propensity of a single particle of a species for each site type(A^1), and a 2-D array for the propensity from a pair of particles for each site type (A^2). If multiple reactions are defined that map to the same indices in these structures, the propensities are added together.

Let A^0_ν denote the zeroth-order propensity for a lattice site of type ν, $A^1_{i,\nu}$ denote the propensity of particle of type i in site ν, and $A^2_{i,j,\nu}$ signifies the $i+j$ particle pair propensity in site ν. Note that A^2 is symmetric, that is, $A^2_{i,j,\nu} = A^2_{j,i,\nu}$, and diagonal entries specify propensities for second-order self dimerizing reactions. The procedure for computing the total propensity for a given lattice site is shown in Algorithm 3.

A particle-based propensity summation benefits from the fact that the number of particles that can be in a lattice site is a fixed quantity. Commonly, it is eight particles to a site. For eight particles per site, algorithm 3 reads at most 37 memory elements. These 37 references are data dependent,

Table I
BENCHMARK MODEL SYSTEMS

Model	Physical Size	Lattice Grid Size	Species	Site Types	Reactions	Initial Particles
Bimolecular	$2\,\mu m \times 2\,\mu m \times 2\,\mu m$	$128 \times 128 \times 128$	2	1	2	$40,000$
Lac Switch	$1\,\mu m \times 1\,\mu m \times 2\,\mu m$	$64 \times 64 \times 128$	12	3	24	$28,936$
Ribosome Biogenesis	$1\,\mu m \times 1\,\mu m \times 4\,\mu m$	$32 \times 32 \times 128$	251	4	884	$37,318$

```
$a_{tot} = A^0_\nu$
$c = |\mathbf{x}_\nu|$
for $i = 0$ to $c$ do
    let $p_i$ = type of the $i^{th}$ particle in $\mathbf{x}_\nu$
    $a_{tot} = a_{tot} + A^1_{p_i,\nu}$
    for $j = i + 1$ to $c$ do
        let $p_j$ = type of the $j^{th}$ particle in $\mathbf{x}_\nu$
        $a_{tot} = a_{tot} + A^2_{p_i,p_j,\nu}$
    end
end
```

Algorithm 3: Procedure for computing the propensity of a given lattice site

Table II
AVERAGE WALL TIME PER TIMESTEP FOR THE REACTION KERNEL USING THE ORIGINAL ALGORITHM AND THE PARTICLE-WISE PROPENSITY TABLES. RUN ON AN NVIDIA TITAN-X GPU.

System	S/RL in Global Mem	Particle-wise
Bimolecular	0.273 ms	0.415 ms
Lac switch	0.264 ms	0.097 ms
Riobsome biogenesis	3.297 ms	0.051 ms

and are essentially random with respect to elements that other threads are reading, resulting in a lot of non-coalesced memory accesses. Recent GPU hardware includes on-chip caches for memory to boost performance for random-access patterns in kernels. Using the read-only data cache path is desirable here as well as it was in reading the RL and S matrices in the original version.

The remainder of the reaction operator kernel is unchanged from the original. A random number is generated to determine if a reaction will occur given the computed total propensity. Then, the list of reactions is linearly scanned to find which reaction occurs. Finally, the S matrix is used to update the particles in the lattice site.

The comparison of the reaction kernel wall-time is shown in Table II. We observe an astounding improvement in performance for the ribosome biogenesis model, over sixty-fold faster than the original. This can be attributed to a few factors. One, the number of memory accesses is considerably lower, as the number of values needed to compute the propensity is independent of the number of reactions. This model also has the smallest lattice of the three systems, which means it requires the smallest number of threads, as one GPU thread computes reactions for exactly one lattice site. The lac switch model improved about 2.75 times, indicating that even modestly complex models can greatly benefit from this approach. However, the bimolecular system ran 1.5 times slower. With only two reactions, the original reaction-oriented approach was preferable.

III. RUN-TIME CODE GENERATION

Starting in CUDA 7.0, the NVIDIA GPU driver can compile CUDA C++ code at runtime. By generating and compiling code at run-time, we have the opportunity to know exactly what reactions are going to be simulated in the model and can emit code that is specific for a set of reactions. In effect, we have the ability to directly embed all of the constant parameters as immediate values in to the source code, instead of fetching them from memory from a data structure. Using the original source code as a template, we can replace sections of the code that read from memory with the actual values that would be stored there. Since we have the specific set of reactions that make up R, we can manually unroll the $r \in R$ loops in algorithm 1. We can emit a series of statements that compute the propensity for a reaction like what is shown in algorithm 2. The value of r_{r1}, r_{r2}, and r_{prop} are all known for each reaction and the value is substituted in the emitted code in place of a variable name. Since r_{order} is known, the entire `if` statement can be replaced with the appropriate propensity function for the reaction.

The same code can be generated for both of the loops over all reactions. However, in the second loop, we need to update the lattice site to reflect the outcome of the reaction. When the reaction to occur is determined, variables are set to the indices of all of the non-zero values of the stoichiometric matrix. We can determine ahead of time what the maximum number of these variables will be needed are and emit appropriate code to handle all of them. As all the values that were stored in constant memory are now directly in the source code, there are no longer any memory restrictions that limit the number of reactions we can simulate at one time, except for the CUDA kernel maximum instruction count.

After generating program code directly from the input model, we can compile the specialized kernel using the nvRTC library. It receives the program source as a string and will return a handle to the resultant binary that can then be executed using the CUDA driver API.

This expectation for this approach is providing constants embedded in code will outperform reading from GPU constant memory. This can be demonstrated with a simple benchmark of four kernels, where each thread iteratively accesses all each element of an array and produces a sum

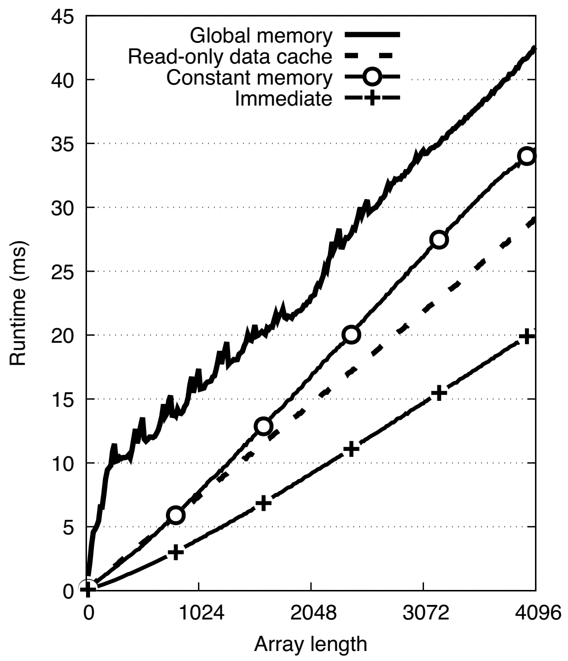

Figure 1. A kernel with immediate values compiled at run-time outperforms kernels utilizing constant memory or global memory when all threads are simultaneously accessing the same element of an array. For access patterns where each thread reads each element in order, utilizing global memory's read-only data cache outperforms using constant memory.

of the result. We will assume for this test a situation where shared memory is otherwise occupied and is unavailable for temporary storage of these arrays. The first kernel reads directly from GPU global memory. The second kernel also uses GPU global memory, but uses the `__ldg` intrinsic to read through the GPU's read-only data cache. The third kernel reads the array elements from constant memory, and the fourth kernel is generated and compiled at run-time to produce the same sequence of arithmetic operations as the other kernels. The first three kernels employ a loop to iterate over the array. To make the comparison more equitable, extra instructions are added to the generated kernel to simulate a loop: incrementing an integer variable as a loop counter and checking the value for terminating the loop. The execution times for these kernels across a range of array lengths can be found in figure 1. GPU global memory performs worse than GPU constant memory, as expected. The difference is most dramatic when accessing a small number of constants. By using the read-only data cache, performance is comparable to using constant memory for small arrays, and outperforms constant memory when reading more than 4 kB (1024 single-precision floating point elements). The best performing kernel is the generated kernel with immediate loads. This asserts that the GPU hardware is well suited to dispatch instructions quickly, and embedding constants as immediate values is viable for improving performance over reading constants from anywhere else in the GPU memory hierarchy.

Table III
AVERAGE WALL TIME PER STEP FOR THE REACTION KERNEL USING THE ORIGINAL ALGORITHM AND RUN-TIME COMPILED CODE ON AN NVIDIA TITAN-X GPU.

System	S/RL in Global Mem	Run-time Compiled
Bimolecular	0.273 ms	0.213 ms
Lac switch	0.264 ms	0.079 ms
Riobsome biogenesis	3.297 ms	5.395 ms

In Table III, the original, compile-time kernel is compared against the run-time generated kernel. The bimolecular switch performance improves with a 1.28 times speedup, which is in contrast to running slower for the particle-wise version. The lac switch improves by 3.3 times, which is greater than the speedup for the particle-wise version. However, the ribosome biogenesis model is now 1.6 times slower, which is remarkably different.

An examination of what went wrong with run-time generation for the ribosome biogenesis model can be found by generating kernels with a subset of the reactions and measuring performance and other metrics. A plot of kernel performance from one to 500 reactions is seen in figure 2. In the lower panel of the figure, the time to execute the kernel (circles) initially shows little variation as reaction count increases. Around 225 reactions, the run time rises sharply. This corresponds directly with the per-thread register count, which hits the maximum value of 255 (lower panel of figure 2, squares). The upper panel of figure 2 shows local memory transaction counts. Local memory, which is GPU global memory that is visible only to a single thread, and is utilized by the compiler for register spills. That is, when the compiler wishes to store intermediate data and there are no registers available, local memory is used to temporarily store information instead. An increase in writing to local memory is coincident with the increase in kernel run time, and is the source of the poor performance.

Before nvRTC was available, we experimented with generating code and including it at run-time to produce a model-specific binary for the ribosome biogenesis project. Using CUDA 5.5 and CUDA 6.5, we did experience better performance over the original kernel, which motivated the effort to utilize nvRTC. We believe that generating code dynamically for a model can be beneficial for even large models, but performance is highly dependent on the compiler version.

IV. CONCLUSION

For both the particle-wise propensity summation and the run-time generated code approaches, the average wall-time for the original reaction kernel was compared against the wall-time for the modified version. Three test models that exhibit a range of complexity were used to evaluate the effectiveness of each approach. The run-time compiled

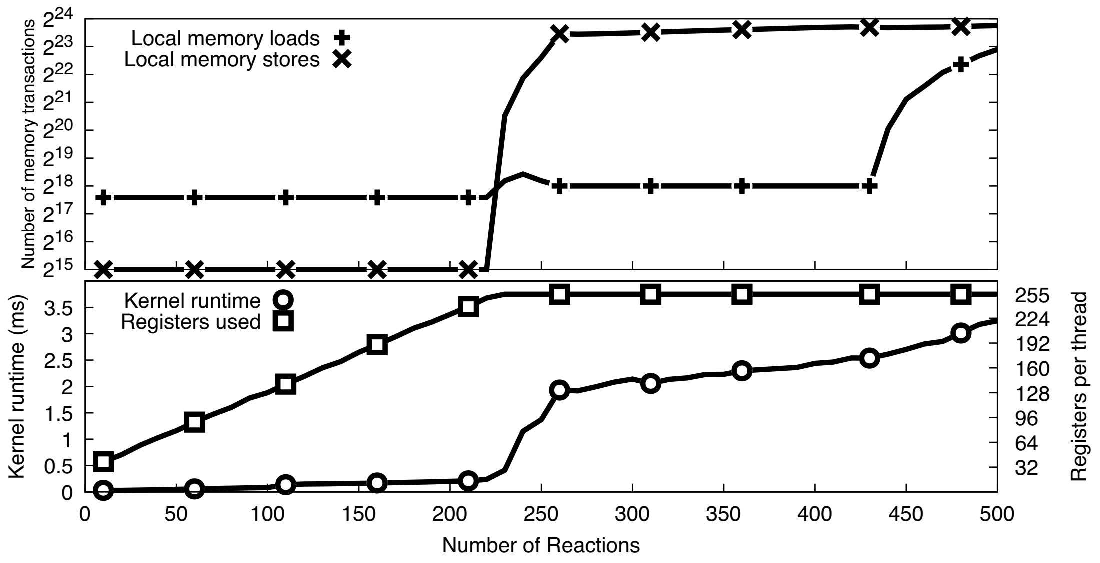

Figure 2. Analysis of reaction kernel runtime, register consumption, and local memory traffic as reaction count grows. Kernel runtime (lower panel, circles) performance suffers once the per-thread register count (lower panel, squares) reaches the maximum of 255. The compiler utilizes register spilling once available registers are exhausted, resulting in a dramatic increase in local memory traffic (upper panel), which hurts performance.

kernel was best for the bimolecular and lac genetic switch models, which would suggest that models with low number of species and reactions are amenable to acceleration via this method. The particle-wise approach is strongly favored for the ribosome biogenesis model, where the high number of reactions are limiting performance.

Run-time code generation is a useful optimization when the compiler is able to keep register utilization down. As the excessive register spilling is the result of attempted optimizations by the compiler, there is hope that a future version of the compiler will behave differently and generate machine code in a manner that makes just-in-time compilation a viable technique for any sized model.

Looking forward, generating GPU code at run-time opens up a new set of features, beyond just improving performance. Similar to OpenMM's custom forces [14], a user could define a propensity calculation that does not have the same format as the rate laws currently supported in MPD-RDME. The two methods for acceleration shown here could also be used in tandem to use a particle-wise method for quickly computing zeroth, first, and second order reactions, and generated code for other types.

V. Acknowledgements

We would like to thank the NIH Center for Macromolecular Modeling and Bioinformatics (NIH-9P41GM104601), the National Science Foundation (Center for the Physics of Living Cells, grant PHY-1430124) and MCB-1244570, for their partial support of this work.

References

[1] S. S. Andrews, N. J. Addy, R. Brent, and A. P. Arkin, "Detailed simulations of cell biology with Smoldyn 21," *PLoS Comput. Biol.*, vol. 6, no. 3, p. e1000705, 2010.

[2] J. Stiles, D. van Helden, T. B. Jr., E. Salpeter, and M. Saltpeter, "Miniature endplate current rise times $< 100\ \mu s$ from improved dual recordings can be modeled with passive acetylcholine diffusion from a synaptic vesicle," *PNAS*, vol. 93, no. 12, pp. 5747–5752, 1996.

[3] J. Schöneberg and F. Noé, "ReaDDy - A Software for Particle-Based Reaction-Diffusion Dynamics in Crowded Cellular Environments," *PLoS ONE*, vol. 8, no. 9, p. e74261, 09 2013. [Online]. Available: http://dx.doi.org/10.1371%2Fjournal.pone.0074261

[4] J. Hattne, D. Fange, and J. Elf, "Stochastic reaction-diffusion simulation with MesoRD," *Bioinform.*, vol. 12, no. 21, pp. 2923–2924, 2005.

[5] E. Roberts, J. E. Stone, and Z. Luthey-Schulten, "Lattice microbes: High-performance stochastic simulation method for the reaction-diffusion master equation," *Journal of Computational Chemistry*, vol. 34, pp. 245–255, 2013.

[6] M. J. Hallock, J. E. Stone, E. Roberts, C. Fry, and Z. Luthey-Schulten, "Simulations of reaction diffusion processes over biologically-relevant size and time scales using multi-gpu workstations," *Parallel Comput.*, vol. 40, pp. 86–99, 2014.

[7] B. Chopard and M. Droz, *Cellular Automata Modeling Of Physical Systems*. Cambridge, UK: Cambridge University Press, 1998.

[8] D. T. Gillespie, "Stochastic simulation of chemical kinetics," *Annu. Rev. Phys. Chem.*, vol. 58, pp. 35–55, 2007.

[9] J. V. Rodríguez, J. A. Kaandorp, M. Dobrzyński, and J. G. Blom, "Spatial stochastic modelling of the phosphoenolpyruvate-dependent phosphotransferase (PTS) pathway in Escherichia coli," *Bioinformatics*, vol. 22, no. 15, pp. 1895–901, 2006.

[10] E. Roberts, J. E. Stone, L. Sepulveda, W. W. Hwu, and Z. Luthey-Schulten, "Long time-scale simulations of *in vivo* diffusion using GPU hardware," in *The Eighth IEEE International Workshop on High-Performance Computational Biology*, May 2009.

[11] T. Earnest, J. Lai, K. Chen, M. Hallock, J. R. Williamson, and Z. Luthey-Schulten, "Towards a whole-cell model of ribosome biogenesis: Kinetic modeling of ssu assembly," *Biophys. J.*, 2015.

[12] J. R. Peterson, M. J. Hallock, E. Roberts, J. A. Cole, P. Labhsetwar, J. E. Stone, and Z. Luthey-Schulten. (2014) Lattice microbes problem solving environment instruction guide. [Online]. Available: http://www.scs.illinois.edu/schulten/lm/download/lm22/InstructionGuide.pdf

[13] NVIDIA Corporation. (2015) Tuning CUDA Applications for Kepler. [Online]. Available: http://docs.nvidia.com/cuda/pdf/Kepler_Tuning_Guide.pdf

[14] P. Eastman, M. S. Friedrichs, J. D. Chodera, R. J. Radmer, C. M. Bruns, J. P. Ku, K. A. Beauchamp, T. J. Lane, L.-P. Wang, D. Shukla, T. Tye, M. Houston, T. Stich, C. Klein, M. R. Shirts, and V. S. Pande, "OpenMM 4: A Reusable, Extensible, Hardware Independent Library for High Performance Molecular Simulation," *Journal of Chemical Theory and Computation*, vol. 9, no. 1, pp. 461–469, 2013, pMID: 23316124. [Online]. Available: http://dx.doi.org/10.1021/ct300857j