Efficient Parallelization of Stochastic Simulation Algorithm for Chemically Reacting Systems on the Graphics Processing Unit *

Hong Li † Linda Petzold ‡

December 1, 2008

*This work was supported in part by the U.S. Department of Energy under DOE award No. DE-FG02-04ER25621, by the National Science Foundation under NSF awards CCF-0428912, CTS-0205584, CCF-326576, and by the Institute for Collaborative Biotechnologies through grant DAAD19-03-D004 from the U.S. Army Research Office.

†Department of Computer Science, University of California, Santa Barbara, CA 93106, U.S.A.

‡Department of Computer Science, Department of Mechanical Engineering, University of California, Santa Barbara, CA 93106, U.S.A.

Proposed running head: Parallelization of SSA on GPU

Author 1: Hong Li

Address: Department of Computer Science,

University of California, Santa Barbara, CA 93106, U.S.A.

Phone number: 805-893-5728 Fax Number: 805-893-5435

Email address: hongli@cs.ucsb.edu

Author 2: Linda Petzold (Corresponding author)

Address: Department of Computer Science,

Department of Mechanical Engineering,

University of California, Santa Barbara, CA 93106, U.S.A.

Phone number: 805-893-5362 Fax number: 805-893-5435

Email address: petzold@engr.ucsb.edu

The special symbols:

```
 \newcommand{\vctr}[1]{\mbox{\boldmath $#1$}}
\newcommand{\timevec}[2]{\vctr{#1}_{#2}}
\newtheorem{example}{\scshape Example}[section]
```

Abstract

The small number of some reactant molecules in biological systems formed by living cells can result in dynamical behavior which cannot be captured by the traditional deterministic approaches. In that case, a more accurate simulation can be obtained with Gillespie's Stochastic Simulation Algorithm (SSA). Since for realistic practical biochemical systems, the simulation by the SSA carries a high computational cost, specifically for large systems, several formulations have been proposed to increase the efficiency of this algorithm. In this paper we propose a highly efficient and scalable formulation of SSA with an optimal static heap structure to achieve logarithmic computational complexity for the whole simulation.

Keywords: Stochastic, SSA, Chemical Reacting Systems, Parallel, GPU

1 Introduction

Chemically reacting systems have traditionally been simulated by solving a set of coupled ordinary differential equations (ODEs). Although the traditional deterministic approaches are sufficient for most systems, they fail to capture the natural stochasticity in some biochemical systems formed by living cells [Gillespie 1976; 1977; McAdams and Arkin 1997; Arkin et al. 1998], in which the small population of a few critical reactant species can cause the behavior of the system to be discrete and stochastic. The dynamics of those systems can be simulated accurately using the machinery of Markov process theory, specifically the stochastic simulation algorithm (SSA) of Gillespie [Gillespie 1976; 1977]. For many realistic biochemical systems the computational cost of simulation by the SSA can be very high. The original form of the SSA is called the Direct Method (DM). Much recent work has focused on speeding up the SSA by reformulating the algorithm [Gibson and Bruck 2000; Cao et al. 2004; McColluma et al. 2005; Blue et al. 1995; Schulze 2002; Li and Petzold 2006].

Often, the SSA is used to generate large (typically ten thousand to a million) ensembles of stochastic realizations to approximate probability density functions of species populations or other output variables. In this case, even the most efficient implementation of the SSA will be very time consuming. Parallel computation on clusters has been used to speed up the simulation of such ensembles [Li et al. 2007]. In [Yoshimi et al. 2005], the use of Field Programmable Gate Arrays (FPGAs) is investigated. However, clusters are still relatively expensive to buy and maintain, and specialized devices such as FPGAs are difficult to program. Due to the low cost and high performance processing capabilities of the GPU, general purpose GPU (GPGPU) computation [GPGPU-Home 2007] has become an active research field with a wide variety of scientific applications including fluid dynamics, molecular dynamics, cellular automata, particle systems, neu-

ral networks, and computational geometry [GPGPU-Home 2007; Li et al. 2008b; Owens et al. 2005; McGraw and Nadar 2007; Li et al. 2008a]. Before the NVIDIA G80 was released, GPU users had to recast their applications into a graphics application programming interface (API) such as OpenGL, which is a significant challenge for non-graphics applications. The Compute Unified Device Architecture (CUDA) release by NVIDIA last year [NVIDIA 2008b] is a technology which directly enables implementation of parallel programs in the C language using an API designed for general-purpose computation. In this paper, we will show how Single Instruction Multiple Data (SIMD) computation can be implemented on a CUDA-enabled GPU, the NVIDIA GeForce 8800GTX, to efficiently perform ensemble runs of SSA simulations for chemically reacting systems.

This paper is organized as follows. In Section 2 we briefly review the Stochastic Simulation Algorithm and some basics of parallel computation with the graphics processing unit. In Section 3 we introduce the efficient parallelization of the SSA on the GPU. Simulation results are presented in Section 4, and in Section 5 we draw some conclusions.

2 Background

2.1 Stochastic Simulation Algorithm

The Stochastic Simulation Algorithm applies to a spatially homogeneous chemically reacting system within a fixed volume at a constant temperature. The system involves N molecular species $\{S_1, \ldots, S_N\}$ represented by the dynamical state vector $X(t) = (X_1(t), \ldots, X_N(t))$ where $X_i(t)$ is the population of species S_i in the system at time t, and M chemical reaction channels $\{R_1, \ldots, R_M\}$. Each reaction channel R_j is characterized by a propensity function a_j and state change vector $\nu_j = \{\nu_{1j}, \ldots, \nu_{Nj}\}$, where $a_j(x)dt$ is the probability, given $X(t) = x$, that

one R_j reaction will occur in the next infinitesimal time interval $[t, t+dt)$, and ν_{ij} is the change in the number of species S_i due to one R_j reaction.

The *Next Reaction Density Function* [Gillespie 2001], which is the basis of SSA, gives the joint probability that reaction R_j will be the next reaction and will occur in the infinitesimal time interval $[t, t+dt)$, given $X(t) = x$. By applying the laws of probability, the joint density function is formulated as follows:

$$P(\tau, j \,|\, \boldsymbol{x}_t, t) = a_j(\boldsymbol{x}_t) e^{-a_0(\boldsymbol{x}_t)\tau}, \tag{1}$$

where $a_0(\boldsymbol{x}_t) = \sum_{j=1}^{M} a_j(\boldsymbol{x}_t)$.

Starting from (1), the time τ, given $X(t) = x$, that the next reaction will fire at $t+\tau$, is the exponentially distributed random variable with mean $\frac{1}{a_0(x)}$,

$$P(\tau|x,t) = a_0(\boldsymbol{x}_t) e^{-a_0(\boldsymbol{x}_t)\tau} \quad (\tau \geq 0). \tag{2}$$

The index j of that firing reaction is the integer random variable with probability

$$P(j|\tau, x, t) = \frac{a_j(\boldsymbol{x}_t)}{a_0(\boldsymbol{x}_t)} \quad (j = 1, ..., M). \tag{3}$$

Thus on each step of the simulation, the random pairs (τ, j) are obtained based on the standard Monte Carlo inversion generating rules: first we produce two uniform random numbers r_1 and r_2 from $U(0,1)$, the uniform distribution on $[0,1]$. Then τ is given by

$$\tau \;\; = \;\; \frac{1}{a_0(\boldsymbol{x}_t)} \ln\left(\frac{1}{r_1}\right). \tag{4}$$

The index j of the selected reaction is the smallest integer in $[1, M]$ such that

$$\sum_{j'=1}^{j} a_{j'}(\boldsymbol{x}_t) > r_2 a_0(\boldsymbol{x}_t). \tag{5}$$

Finally, the population vector X is updated by the state change vector ν, and the simulation is advanced to the next reacting time.

Because SSA must simulate every reaction event, simulation with SSA can be quite computationally demanding. A number of different formulations of SSA have been proposed, in an effort to speed up the simulation [Gibson and Bruck 2000; Cao et al. 2004; McColluma et al. 2005; Blue et al. 1995; Schulze 2002; Li and Petzold 2006]. The most time-consuming step of the SSA is the selection of the next reaction to fire. The complexity of this step for the Direct Method is $O(M)$, where M is the number of reactions. To the best of our knowledge, the fastest known SSA formulation is something we call the Logarithmic Direct Method (LDM) because its complexity for the critical step is $O(logM)$. The LDM algorithm comes from the literature on Kinetic Monte Carlo (KMC) algorithms[Schulze 2002]. The SSA is a type of KMC algorithm that is applied to chemical kinetics. Because of the special structure of the chemical kinetics problems, it has been possible to put SSA on a solid theoretical foundation. Further efficiency of the LDM can be achieved by using sparse matrix techniques in the system state update stage [Li and Petzold 2006]. In our performance comparisons, we use the LDM with sparse matrix update. The algorithm is summarized as follows:

1. *Initialization*: Initialize the system.

2. *Propensity calculation*: Calculate the propensity functions a_i $(i = 1, ..., M)$, and save the intermediate data as an ordered sequence of the propensities subtotalled from 1 to M, while summing all the propensity functions to

obtain a_0 .

3. *Reaction time generation*: Generate the firing time of the next reaction.

4. *Reaction selection*: Select the reaction to fire next with binary search on the ordered subtotal sequence.

5. *System state update*: Update the state vector x by ν_j with sparse matrix techniques, where j is the index of the current firing reaction. Update the simulation time.

6. *Termination*: Go back to stage 2 if the simulation has not reached the desired final time.

When an ensemble (ten thousand to a million realizations or more) must be generated, the computation can become intractable even with the best SSA formulation. Thus we seek to make use of the low-cost, high efficiency GPGPU.

2.2 Using the Graphics Processor Unit as a Data Parallel Computing Device

Modern Graphics Processor Unit

The Graphics Processing Unit (GPU) is a dedicated graphics card for personal computers, workstations or video game consoles. Recently, GPUs with general purpose parallel programming capacities have become available. The GPU has a highly parallel structure with high memory bandwidth and more transistors devoted to data processing than to data caching and flow control (compared with a CPU architecture), as shown in Figure 1 [NVIDIA 2008a]. This makes the GPGPU a very powerful computing engine. NVIDIA reports that the GPU architecture is most effective for problems that can be implemented with stream

processing and using limited memory. Single Instruction Multiple Data (SIMD), which involves a large number of totally independent records being processed by the same sequence of operations simultaneously, is an ideal general purpose graphics processing unit (GPGPU) application.

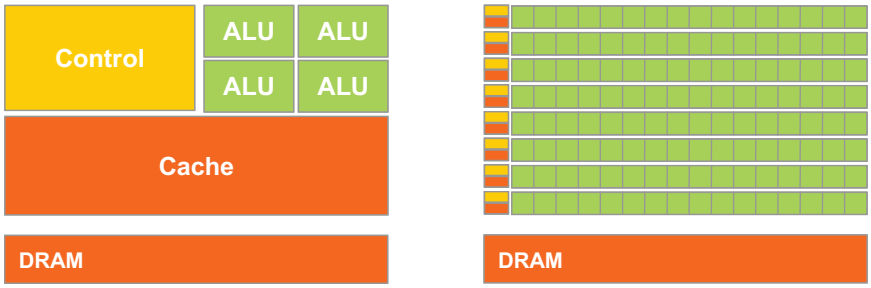

Figure 1:

NVIDIA 8 Series GeForce-based GPU Architecture

NVIDIA corporation claims its graphics processing unit (GPU) as a "second processor in personal computers" [NVIDIA 2008b], which means that the data parallel computation intensive part of applications can be off-loaded to the GPU [NVIDIA 2008a].

We performed our simulations on the NVIDIA 8800 GTX chip with 768MB RAM. There are 128 stream processors on a $480mm^2$ surface area of the chip, divided into 16 clusters of multiprocessors as shown in Figure 2 [NVIDIA 2008a]. Each multiprocessor has 16 KB shared memory which brings data closer to the ALU. The processors are clocked at 1.35 GHz with dual processing of scalar operations supported. Thus the peak computation rate accessible from the CUDA is (16 multiprocessors * 8 processors / multiprocessor) * (2 flops / MAD) [1] * (1 MAD / processor-cycle) * 1.35 GHz = 345.6 GFLOP/s. The maximum observed bandwidth between system and device memory is about 2GB/second. All of the

[1] A MAD is a multiply-add.

benchmarks on the GPU were performed on a single Geforce 8800 GTX GPU card. Likewise, we use only use a single core of the Intel Core 2 Duo E6700 2.67GHz dual-core processor [PCperspective 2008], which makes the best use of the memory bandwidth.

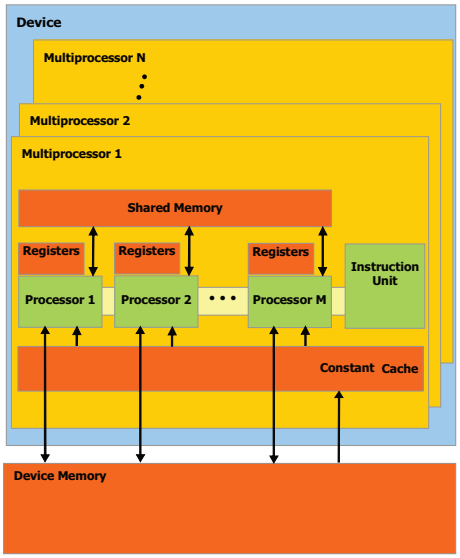

Figure 2:

The limited size of the shared memory of each multiprocessor restricts the range of applications that can make use of this architecture. Maximizing the use of shared memory makes better use of the arithmetic units. The Compute Unified Device Architecture (CUDA) Software Development Kit (SDK), supported by the NVIDIA Geforce 8 Series makes this challenging task easier than previous graphics APIs.

CUDA: A GPU Software Development Environment

The CUDA provides an essential high-level development environment with standard C language, resulting in a minimal learning curve for beginners to access the low-level hardware. Unlike previous graphics application interfaces, the CUDA

provides both scatter and gather memory operations for development flexibility. It also supports fast read and write shared memory to reduce the dependence of application performance on the DRAM bandwidth [NVIDIA 2008a].

The structure of CUDA computation broadly follows the data-parallel model: each of the processors executes the same sequence of instructions on different sets of the data in parallel. The data can be broken into a 1D or 2D grid of blocks, and each block can be 1D, 2D or 3D and can allow up to 512 threads which can collaborate through shared memory. Threads within a block can collaborate via the shared memory. Currently the multiprocessor single-instruction multiple-thread unit manages threads in warps, which is 32 parallel threads. Threads in a warp execute the same instructions at a time, thus the branch divergence in one warp will cause each branch path execute serially on each path. Since different warps run independently, to fully utilize the GPU, we should try to make the threads in a warp take the same execution path[NVIDIA 2008a].

In theory, the CPU and GPU can run in parallel. In practice, the severe memory limitations of the G80 makes this impossible for all but the smallest problems. The problem is that, if we have two kernels, $K1$ and $K2$, one of which is running on the (single) GPU, then in order to transfer the data needed by K2 into the GPU memory while $K1$ is simultaneously executing, one would need to partition the already small GPU device memory into parts. This puts a very server restriction on the amount of memory available to each kernel.

3 Implementation Details

3.1 Parallelism across the simulations

NVIDIA reported that streaming processing, which allows many applications to more easily exploit a limited form of parallel processing [Encyclopedia 2008], can

run very efficiently on the new GPU architecture. Our focus is on computation of ensembles of SSA realizations, which is a typical stream processing application. Ensembles of SSA simulations for chemically reacting systems are very well-suited for implementation on the GPU through the CUDA. The simulation code can be put into a single kernel running in parallel on a large set of system stat vectors $X(t)$. The large set of final stat vectors $X(t_{final})$ will contain the desired results.

The initial conditions $X(0)$ and the stoichiometric matrix ν originally will be in the host memory. We must copy them to the device memory by CUDAMemcpy in the driver running on the the CPU. We minimize the transfer between the host and device by using an intermediate data structure on the device and batch a few small transfers into a big transfer to reduce the overhead for each transfer. Next, we need to consider the relatively large global memory vs. the limited-size shared memory. The global memory adjacent to the GPU chip has higher latency and lower bandwidth than the on-chip shared memory. There is about 400-600 clock cycle latency to access the global memory vs. 4 clock cycles to read or write the shared memory. To effectively use the GPU, our simulation makes as much use of on-chip shared memory as possible. We load $X(0)$ and the stoichiometric matrix ν from the device memory to the shared memory at the very beginning of the kernel, process the data (propensity calculation, state vector update, etc.) in shared memory, and write the result back to the device memory at the end. Because the same instruction sequence is executed for each data set, there is a low requirement for flow control. This matches the GPU's architecture. The instruction sequence is performed on a large number of data sets which do not need to swap out, hence the memory access latency is negligible compared with the arithmetic calculation.

The CUDA allows each block to contain at most 512 threads, but blocks with

the same dimension and size that run the same kernel can be put into a grid of blocks. Thus the total number of threads for one kernel can be very large. Given the total number of realizations of SSA to be simulated, the number of threads per block and the number of blocks must be carefully balanced to maximize the utilization of computation resources. Otherwise, the it will cause the uncoalesced addressing which will slow down the simulation. For stochastic simulation, we can't use too many threads per block since there is only a limited shared memory and all system state vectors and propensities have been put in shared memory for efficient frequent access. Thus the number P of threads per block should satisfy $(N+M)*4*P+\alpha < 16K$, where N is the number of chemical species, M is the number of reactions, 4 is the size (in bytes) of an integer/float variable, $16K$ is the maximum shared memory we can use within one block, and α is the shared memory used by the random number generator (this is relatively small).

Random Number Generation

Statistical results can only be relied on if the independence of the random number samples can be guaranteed. Thus generating independent sequence of random numbers is one of the important issues of implementing simulation for ensembles of stochastic simulation algorithms in parallel.

Originally we considered pre-generating a large number of random numbers by the CPU. Since the CPU and GPU can't communicate in real time in parallel (the GPU has to stop to get the data from the CPU and then continue the computation), we can pre-generate a huge number of random numbers and store them in the shared memory and swap back to the CPU to generate more when they are used up. Alternatively, we could pre-generate a huge number of random numbers and put them in the global memory. Both methods will waste too much time for data access. Furthermore, the Scalable Parallel Random Number

Generators Library (SPRNG) [Mascagni 1999; Mascagni and Srinivasan. 2000], which we use in our StochKit [Li et al. 2007] package for discrete stochastic simulation because of its excellent statistical properties, cannot be implemented on the GPU due to its complicated data structure. The only solution appears to be to implement a simple random number generator on the GPU. Experts suggest using a mature random number generator instead of inventing a new one, since it requires great care and extensive testing to evaluate a random number generator[Brent 1992]. Thus we chose the Mersenne Twister from the literature in our application [Forums Members 2008].

The Mersenne Twister (MT) was developed by Makoto Matsumoto and Takuji Nishimura [Matsumoto and Nishimura 1998] in 1997, with initialization improved in 2002 [Podlozhnyuk 2008]. This method has passed many statistical randomness tests including the stringent Diehard tests [Forums Members 2008]. The fully tested MT random number generator can efficiently generate high quality, long period random sequences with high order of dimensional equidistribution. Another good property of the MT is its efficient use of memory. Thus it is very suitable for our application. In our implementation we modified Eric Mills's multithreaded C implementation [Forums Members 2008]. Since our application requires a huge number of random numbers even for one realization of a simple model, we use the shared memory for random number generation to minimize the data launching and accessing time.

4 Parallel Simulation Performance

The performance of the parallel simulation is limited by the number of processors available for the computation, the workload of the available processors, and the communication and synchronization costs. It is important to note that more

processors does not necessarily mean better performance. Our simulations were run on a single NVIDIA GeForce 8800GTX GPU installed on a personal workstation. The benchmarking on the GPU has been done on a configuration consisting of the host workstation and one GPU card. Likewise the benchmarking on the CPU was performed on a single core. For the benchmarking on the CPU, we compiled the code without and with the SSE extension, where we generate the SSE code automatically via the compiler without hand-coded assembly. In our tests, both the GPU and the CPU simulations were done in single precision, because the G80 supports only single precision. One might legitimately wonder to what extent this impacts the accuracy of the computation. To this end, we performed both experiments in double precision on the CPU, and found that the difference between the single precision and the double precision results was not statistically significant.

Example 4.1 Decay Dimerization Model

The decay dimerization model [Gillespie 2001] involves three reacting species S_1, S_2, S_3 *and four reaction channels* R_1, R_2, R_3, R_4

$$\begin{aligned} S_1 &\xrightarrow{c_1} 0 \\ S_1 + S_1 &\underset{c_2}{\overset{c_3}{\rightleftharpoons}} S_2 \\ S_2 &\xrightarrow{c_4} S_3. \end{aligned} \tag{6}$$

We used the reaction rate constants from [Gillespie 2001],

$$c_1 = 1,\ c_2 = 0.002,\ c_3 = 0.5,\ c_4 = 0.04, \tag{7}$$

and the initial conditions

$$X_1 = 10^4,\ X_2 = X_3 = 0. \tag{8}$$

The simulation performance has been extraordinary, as shown in Table 1. For 30 000 *realizations, the parallel (GPU) simulation is almost* 200 *times faster than the sequential simulation on the host computer.* [2]

In general, as the size of the system increases, the speed-up of the GPU decreases because of the limited shared memory. However, most biochemical systems are loosely coupled, thus we can make use of sparse matrix techniques to reduce the memory requirements. Here we use the Yale Sparse Matrix Format[Encyclopedia 2008]. Very large biochemical systems arise when the model takes into account spatial inhomogeneity. The SSA is based on the assumption of a spatially homogeneous system. However, by discretizing the space into cells and introducing variables associated to the population of the species in each cell, the SSA can also be applied to spatially inhomogeneous systems. Here we construct a simple example to illustrate the power of the GPU for this type of problem.

Example 4.2 Spatially Inhomogeneous Model

This model was introduced by Shnerb et al. [Shnerb et al. 2000] to illustrate the difference between the continuous deterministic approach and the discrete stochastic approach. For convenience, we simplified the model slightly by fixing the position of one species. The model is defined on a 2-dimensional grid. Species A is initially located at a single grid point and moves randomly with a given diffusion coefficient. Species B is initially located in a randomly-chosen area of adjacent grid points, away from the border regions. The model is simulated over a fixed time period, to find the spatial distribution of A. Two types of reactions are involved. Species A decays with a constant rate μ, and divides with rate λ

[2]We note that the compiler generated SSE code did not yield much improvement. This may be due to factors such as the high degree of data dependence from one step to another, unexpected loop exits involved in the determination of which reaction will fire first, and non-sequential data accesses due to the sparse structure of the network stoichiometric matrix.

when it meets the catalyst B. We simulated the model with $n = 8, 10, 16, 20$. To each grid cell (labeled (i, j)), we assign variables $A_{i,j}$ for species A and $B_{i,j}$ for species B. The reactions are listed as follows

$$\begin{aligned} &\textit{Diffusion of } A: && A_{i,j} \longrightarrow A_{i\pm1,j\pm1}, \\ &\textit{Decay of } A: && A_{i,j} \longrightarrow \emptyset, \\ &\textit{Division of } A: && B_{i,j} + A_{i,j} \longrightarrow B_{i,j} + 2A_{i,j} \end{aligned} \tag{9}$$

The diffusion rate for A is $\mu = 0.5$. The decay rate for A is 0.1. The division rate (when A reaches the region occupied by B) is $\lambda = 0.0025$. The initial states are set so that one cell contains a population of 100 of species A, 4 cells contain a population of 100 of species B, and the remainder of the cells contain no A or B respectively.

$$\begin{aligned} A_{i,j} &= \begin{cases} 100, & i = 10, j = 10, \\ 0, & else. \end{cases} \\ B_{i,j} &= \begin{cases} 100, & selected\ i, j, \\ 0, & else. \end{cases} \end{aligned} \tag{10}$$

To use the shared memory efficiently, in addition to using the sparse matrix technique we group the M reactions according to the type of the reaction. We first determine which group will fire next, and then determine at which grid point that type of reaction will fire. By doing this, we can avoid saving the propensities at each grid point, which is what is normally done in SSA [Cao et al. 2004; Li and Petzold 2006]. Instead, we keep track only of the number of species A at each grid point. By doing this, we can dramatically reduce the number of operations consumed in the calculation of the propensities, as well as the use of the shared memory and global memory. The disadvantage is that we must update the propensities for each group very frequently. We measured the CPU time for 40 000 *realizations. The timing results for different grids are shown in Table*

2. The parallel (GPU) simulation is about 200 *times faster than the sequential simulation on the host computer.*

For these computations, we have been able to store all frequently-used reaction rates in shared memory. Because the shared memory is limited, it is not possible to store all of the data for a large grid like $100 * 100$ *in shared memory. Additionally, we can't run too many realizations for a large grid at the same time, because the device memory of the GeForce 8800 GTX is also limited (768M). For* 5 000 *realizations of the* $100 * 100$ *grid, the parallel simulation is about 50 times faster than the sequential one.*

5 Conclusions

The SSA is the workhorse algorithm for discrete stochastic simulation in systems biology. Even the most efficient implementations of the SSA can be very time-consuming. Often the SSA is used to generate ensembles (typically ten thousand to a million) of stochastic simulations. The current generation of GPUs appears to be very well-suited for this purpose. On the two model problems we tested, we observed speedups about 200 times for the GPU, over the time to compute on the host workstation. With this impressive performance improvement, in one day we can generate data which would require more than six months of computation with the sequential code.

This technology is not quite ready for the novice user. Programs must be written to be memory efficient, with the GPU architecture in mind.

List of Captions:

References

Arkin, A., J. Ross, and H. McAdams (1998). Stochastic kinetic analysis of developmental pathway bifurcation in phage λ-infected E. Coli cells. *Genetics 149*, 1633–1648.

Blue, J., I. Beichl, and F. Sullivan (1995). Faster Monte Carlo simulations. *Physical Rev. E 51*, 867–868.

Brent, R. P. (1992). *Report TR-CS-92-02* .

Cao, Y., H. Li, and L. Petzold (2004). Efficient formulation of the stochastic simulation algorithm for chemically reacting systems. *J. Phys. Chem. 121*(9), 4059–4067.

Encyclopedia, T. F. (2008). Wikipedia. http://en.wikipedia.org.

Forums Members, N. F. (2008). NVIDIA forums. http://forums.nvidia.com.

Gibson, M. and J. Bruck (2000). Efficient exact stochastic simulation of chemical systems with many species and many channels. *J. Phys. Chem. 105*, 1876–1889.

Gillespie, D. (1976). A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *J. Comp. Phys. 22*, 403–434.

Gillespie, D. (1977). Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem. 81*, 2340–2361.

Gillespie, D. (2001). Approximate accelerated stochastic simulation of chemically reacting systems. *J. Chem. Phys. 115*(4), 1716–1733.

GPGPU-Home (2007). GPGPU homepage. http://www.gpgpu.org/.

Li, H., Y. Cao, L. Petzold, and D. Gillespie (2007). Algorithms and software for stochastic simulation of biochemical reacting systems. *Biotechnology Progress. 24*, 56–61.

Li, H., A. Kolpas, L. Petzold, and J. Moehlis (2008a). Efficient parallel simulation of an individual-based fish schooling model on a graphics processing unit. In *Grace Hopper Celebration of Women in Computing*.

Li, H., A. Kolpas, L. Petzold, and J. Moehlis (2008b). *Concurrency and Computation: Practice and Experience* . to appear.

Li, H. and L. Petzold (2006). Logarithmic Direct Method for discrete stochastic simulation of chemically reacting systems. Technical report, Department of Computer Science, University of California, Santa Barbara. http://www.engr.ucsb.edu/∼cse.

Mascagni, M. (1999). SPRNG: A scalable library for pseudorandom number generation. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas.

Mascagni, M. and A. Srinivasan. (2000). SPRNG: A scalable library for pseudorandom number generation. In *ACM Transactions on Mathematical Software*, Volume 26, pp. 436–461.

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

McAdams, H. and A. Arkin (1997). Stochastic mechanisms in gene expression. *Proc. Natl. Acad. Sci. USA 94*, 814–819.

McColluma, J. M., G. D. Peterson, C. D. Cox, M. L. Simpson, and N. F. Samatova (Feb. 2005). The sorting direct method for stochastic simulation of biochemical systems with varying reaction execution behavior. *J. Comput. Biol. Chem. 30*, 39–49.

McGraw, T. and M. Nadar (2007). Stochastic DT-MRI connectivity mapping on the gpu. *IEEE Transactions on Visualization and Computer Graphics 13*(6), 1504–1511.

NVIDIA (2008a). NVIDIA CUDA Compute Unified Device Architecture Programming Guide. http://developer.download.nvidia.com.

NVIDIA (2008b). NVIDIA homepage. http://www.nvidia.com.

Owens, J. D., D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell (2005, August). A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pp. 21–51.

PCperspective (2008). PCperspective. http://www.pcper.com.

Podlozhnyuk, V. (2008). Mersenne Twister. http://developer.download.nvidia.com.

Schulze, T. P. (2002). Kinetic Monte Carlo simulations with minimal searching. *Physical Review E 65*(3), 036704.

Shnerb, N., Y. Louzoun, E. Bettelheim, and S. Solomon (2000). The Importance of Being Discrete - Life Always Wins on the Surface Proc. *Proc. Natl. Acad. Sci. USA 97*, 10332.

Yoshimi, M., Y. Osana, Y. lwaoka, A. Funahashi, N-Hiroi, Y. Shibata, N. lwanaga, H. Kitano, and H. Amano (2005). The design of scalable stochas-

tic biochemical simulator on FPGA. *Proc. of I. C. on Field Programmable Technologies (FPT2005)* , 139–140.

List of Tables:

Table 1: Performance for Dimer Decay model

T*B	R	ST	STsse	PT	GGPU
16*16	256	11.6065	11.3201	0.6354	4.3968
16*32	512	23.2192	22.9833	0.6655	8.3978
32*32	1024	46.4077	46.1381	0.6789	16.4556
64*32	2048	92.8379	92.0769	0.7354	30.3889
128*32	4096	185.5898	184.9292	0.9984	44.7435
256*32	8192	371.2942	370.8793	1.8462	48.4103
256*64	16 384	742.8669	742.1035	3.6357	49.1821
256*96	24 578	1 114.1775	1 113.7827	5.2921	50.6778
256*128	32 768	1 477.8368	1 476.4513	6.8798	51.7059

This table shows the performance for Dimer Decay model, where $T * B$ is the thread number $*$ block number, R is the number of realizations, ST is the sequential simulation time, $STsse$ is the simulation time on the CPU with the SSE extension, PT is the parallel simulation time, and $GGPU$ is the GFLOPS on the GPU.

Table 2: Performance for the Spatially Inhomogeneous model

S	N	M	ST	STsse	PT	GGPU
8*8	68	388	127.2968	118.3672	0.5068	81.2168
10*10	104	604	204.0062	192.3844	0.8563	77.0137
16*16	260	1 536	511.4982	487.9421	2.3386	70.7090
20*20	404	2 404	785.6463	757.4982	3.8153	66.5706

This table shows the performance for the Spatially Inhomogeneous model, where S is the system size, N is the number of species, M is the number of reactions, ST is the sequential simulation time, $STsse$ is the simulation time on the CPU with the SSE extension, PT is the parallel simulation time, and $GGPU$ is the GFLOPS on the GPU.