EFFICIENT PARALLELIZATION OF THE STOCHASTIC SIMULATION ALGORITHM FOR CHEMICALLY REACTING SYSTEMS ON THE GRAPHICS PROCESSING UNIT

Hong Li
Linda Petzold

DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF CALIFORNIA, SANTA BARBARA, CA 93106, U.S.A. (PETZOLD@CS.UCSB.EDU)

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 traditional deterministic models. In such a problem, a more accurate simulation can be obtained with discrete stochastic simulation (Gillespie's stochastic simulation algorithm – SSA). Many stochastic realizations are required to capture accurate statistical information of the solution. This carries a very high computational cost. The current generation of graphics processing units (GPU) is well-suited to this task. In this paper we describe our implementation and present some computational experiments illustrating the power of this technology for this important and challenging class of problems.

Key words: stochastic, SSA, chemically 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, Ross, and McAdams 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 (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 (Blue, Beichl, and Sullivan 1995; Gibson and Bruck 2000; Schulze 2002; Cao, Li, and Petzold 2004; McColluma et al. 2005; 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). Yoshimi et al. (2005) investigated the use of field programmable gate arrays (FPGAs). However, clusters are still relatively expensive to buy and maintain, and specialized devices such as FPGAs are difficult to program. Because of the low cost and high performance processing capabilities of the GPU, general purpose GPU (GPGPU) computation (GPGPU 2007) has become an active research field with a wide variety of scientific applications including fluid dynamics, molecular dynamics, cellular automata, particle systems, neural networks, and computational geometry (Owens et al. 2005; GPGPU 2007; McGraw and Nadar 2007; Li et al. 2008, 2009). 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; NVIDIA 2008a) is a parallel computing architecture which unlocks the computational power of the GPU to scientific computing through APIs designed for general-purpose computation in a C-like language. In this paper, we will show how to efficiently perform ensemble runs of SSA simulations for chemically reacting systems on a CUDA-enabled GPU – the NVIDIA GeForce 8800GTX – and demonstrate that very substantial speedups are achievable even for large problems that do not fit in the shared memory.

The International Journal of High Performance Computing Applications,
Volume 24, No. 2, Summer 2010, pp. 107–116
DOI: 10.1177/1094342009106066

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 as a result of 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:

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

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

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 $\frac{1}{a_0(x)}$,

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

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

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

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

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

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

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

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

The SSA is a type of kinetic Monte Carlo (KMC) algorithm that is applied to chemical kinetics. Because of the special structure of chemical kinetics problems it has been possible to put SSA on a solid theoretical foundation. 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 (Blue et al. 1995; Gibson and Bruck 2000; Schulze 2002; Cao et al. 2004; McColluma et al. 2005; 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. A fast 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 algorithms (Schulze 2002). 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 ($i = 1, \ldots, M$), and save the intermediate data as an ordered sequence of the propensities subtotaled 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

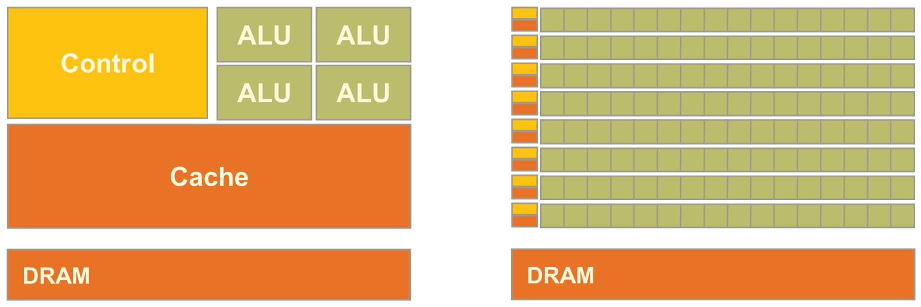

Fig. 1 CPU (left) versus GPU (right) architecture.

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

2.2.1 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 GPGPU application.

2.2.2 NVIDIA 8 Series GeForce-based GPU architecture NVIDIA corporation claims its GPU as a "second processor in personal computers," 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 768 MB RAM. There are 128 stream processors on a 480 mm^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 Arithmetic Logic Unit (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/multiproc-

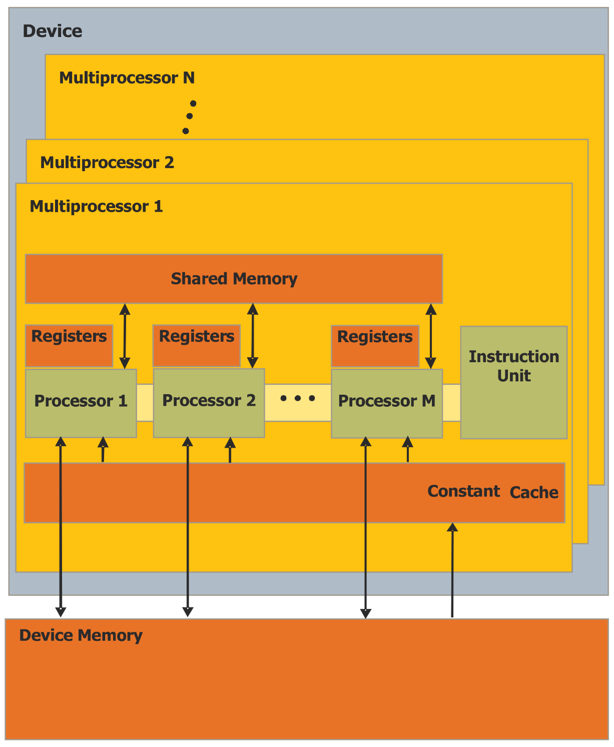

Fig. 2 Hardware model.

essor) × (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 2 GB/s. All of the benchmarks on the GPU were performed on a single GeForce 8800 GTX GPU card. Likewise, we use only a single core of the Intel Core 2 Duo E6700 2.67 GHz dual-core processor, which makes the best use of the memory bandwidth.

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.

2.2.3 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 1-D or 2-D grid of blocks, and each block can be 1-D, 2-D or 3-D and can allow up to 512 threads which can collaborate through shared memory. Currently the multiprocessor single-instruction multiple-thread unit manages threads in warps comprising 32 parallel threads. Threads in a warp execute the same instructions at one time, thus the branch divergence in one warp will cause each branch path execute serially. Because 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 severe restriction on the amount of memory available to each kernel.

3 Implementation Details

3.1 Random Number Generation

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

Originally we considered pre-generating a large number of random numbers by the CPU. Because the CPU and GPU cannot 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 because of its complicated data structure. The only solution appears to be to implement a simple random number generator (RNG) on the GPU. Experts suggest using a mature random number generator instead of inventing a new one, because it requires great care and extensive testing to evaluate a random number generator (Brent 1992). Thus for our simulation we chose the Mersenne Twister (Matsumoto and Nishimura 1998; Podlozhnyuk 2008), which has been designed to address the flaws of previous RNGs and is now widely used.

The Mersenne Twister (MT), was developed by Makoto Matsumoto and Takuji Nishimura in 1997 (Matsumoto and Nishimura 1998), with initialization improved in 2002 (Matsumoto and Nishimura 2002). The MT pseudorandom number generator is based on a matrix linear recurrence over a finite binary field and generates the vectors of fixed word size:

$$x_{(k+n)} := x_{(k+m)} + (x_k^u | x_{(k+1)}^l) \cdot A (k = 0, 1, \ldots) \qquad (6)$$

where, n, m are fixed positive integers and $n > m$; $(x_k)_{(k=0,1,\ldots)}$ are a sequence of vectors with fixed width; $(x_k^u | x_{(k+1)}^l)$ is the w-dimensional vector with the concatenation of the $w - r$ most significant of bits of x_k and the r least significant bits of $x_{(k+1)}$, w is the word size and r is the separation point; and $(x_k^u | x_{(k+1)}^l) \cdot A$ is the multiplication of the concatenated bit vector with the matrix A

(called the twister); The form of the matrix A is chosen for quick computation (A is a companion matrix):

$$A = \begin{pmatrix} & 1 & & & & \\ & & 1 & & & \\ & & & \cdot & & \\ & & & & \cdot & \\ & & & & & \cdot \\ & & & & & 1 \\ a_0 & a_1 & \cdot & \cdot & \cdot & a_{(w-1)} \end{pmatrix}. \tag{7}$$

For such an A, the multiplication $x \cdot A$ can be computed by

$$x \cdot A = \begin{cases} \text{shiftright}(x), \\ \quad \text{if the least significant bit of } x \text{ is } 0, \\ \text{shiftright}(x) \oplus a, \text{ else} \end{cases} \tag{8}$$

where $a = (a_0, a_1, \ldots, a_{(w-1)})$.

This method has passed many statistical randomness tests including the stringent Diehard tests (Matsumoto and Nishimura 1998). The fully tested sequential MT random number generator can efficiently generate high quality, long period random sequences with a high order of dimensional equidistribution. Because of MT's bitwise arithmetic and efficient use of memory, the algorithm is well-suited to the GPU with CUDA. In our implementation, we use the Mersenne Twister MT19937 with 32-bit word length on the GPU, the modification of the multithreaded C implementation of MT for GPU supplied by NVIDIA SDK (NVIDIA 2008b). Because of the iterative nature of the MT, the limitations of the GPU memory access and the many random numbers needed simultaneously for each launch of our simulation, it is hard to parallelize a single MT on the GPU. The generation of a large number of independent parallel streams of random numbers is known to be a difficult problem that is best addressed by experts. Thus we are using the Dynamic Creation of Pseudorandom Number Generators (DCPRNG) proposed by MT authors M. Matsumoto and T. Nishimura in 2000. This method is based on a hypothesis that many PRNG researchers agree with: "A set of PRNGs based on linear recurrences is mutually `independent' if the characteristic polynomials are relatively prime to each other" (Matsumoto and Nishimura 2000). Since our application requires a huge number of random numbers for even one realization of a simple model, we put the state vector in the shared memory for random number generation, to minimize the data launching and accessing time.

3.2 Parallelism Across the Simulations

NVIDIA reported that stream processing, which allows many applications to more easily exploit a limited form of parallel processing, 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 state vectors $X(t)$. The large set of final state 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 versus 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 a 400–600 clock cycle latency to access the global memory compared with four 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. In addition, the global memory is not cached, so it is important to choose the right memory access pattern to achieve maximum bandwidth, which means we need to align addresses to 4, 8, and 16 to avoid an uncoalesced addressing problem. For the stochastic simulation of biochemically reacting systems, we utilize a fixed number of threads per block to efficiently use the limited shared memory. For the best performance, and whenever possible, all system state vectors and propensities should be stored in shared memory for efficient frequent access. To put all system state and propensities in the shared memory, the number P of threads per block should satisfy $(N + M) \times 4 \times P + \alpha < 16$ K, 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, 16 K 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). For those large systems for which it is impossible to fit all the data in the limited shared memory for many threads, we store only the most frequently used data there, to maximize possible calculation before switching them out. To run a large number of realizations, we launch many blocks simultaneously. Up to a point, the performance increases as the number of blocks increases. This is because the different warps running on the GPU in turn can hide the memory latency. But when the number of threads is large enough to hide the memory latency, further increase in the number of threads will slow down the performance.

In our experiments, we measure the histogram distance (Cao and Petzold 2006) between the two sets of data for each species computed on the GPU and the CPU. Suppose I is the interval that contains all the sample values, $I = [x_{min}, x_{max})$, and $L = x_{max} - x_{min}$. The interval I is divided into K subintervals $I_i = [x_{min} + \frac{(i-1)L}{K}, x_{min} + \frac{iL}{K})$, $_{(i = 1, 2, \ldots K)}$. The histogram distance between two sets X_i and Y_j of samples is defined as

$$D_K(X, Y) = \sum_{i=1}^{K} \left| \frac{\sum_{j=1}^{N} \chi(x_j, I_i)}{N} - \frac{\sum_{j=1}^{M} \chi(y_j, I_i)}{M} \right|, \tag{9}$$

where $\chi(x, I_j)$ is the characteristic function defined as

$$\chi(x, I_j) = \begin{cases} 1, & \text{if } x \in I_i, \\ 0, & \text{else.} \end{cases} \tag{10}$$

To calculate the histogram distance, we first determine the *min* and *max* values, collect all the points into the bins and get a raw count of the bin population. Then we normalize the bin population to a percentage of the points that lie in that bin (Cao and Petzold 2006). On the GPU, we use a tree-based approach to do the parallel reduction within each block. Between blocks, we switch back to the CPU for each step of the reduction, to get the *min* and *max*. To get the percentage of normalized samples of the interested species that fall in each bin's interval, each thread processes its own species population and adds itself to the appropriate place in the bin vector. The histogram distance between the GPU and the CPU ensemble simulations is then compared with the self-distance of the CPU simulation (the distribution distance of two ensemble runs done with different random number seeds) to determine whether they are statistically significant. All such ensemble simulations are comprised of the same number of stochastic realizations, that is, we choose $N = M$.

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 (Cao and Petzold 2006) for 100,000 realizations. (The self-distance depends on the number of realizations.)

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

$$\begin{aligned} &S_1 \xrightarrow{c_1} 0 \\ &S_1 + S_1 \underset{c_2}{\overset{c_3}{\rightleftharpoons}} S_2 \\ &S_2 \xrightarrow{c_4} S_3. \end{aligned} \tag{11}$$

We used the reaction rate constants from Gillespie (2001),

$$c_1 = 1, \quad c_2 = 0.002, \quad c_3 = 0.5, \quad c_4 = 0.04, \tag{12}$$

Table 1
Performance for the 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	1,024	46.4077	46.1381	0.6789	16.4556
64 × 32	2,048	92.8379	92.0769	0.7354	30.3889
128 × 32	4,096	185.5898	184.9292	0.9984	44.7435
256 × 32	8,192	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 the 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.

and the initial conditions

$$X_1 = 10^5, \quad X_2 = X_3 = 0. \tag{13}$$

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, 3] □

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 (Golub and Loan 1996). 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. (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 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

$$\begin{aligned} &\text{Diffusion of A:} && A_{i,j} \rightarrow A_{i\pm 1, j\pm 1}, \\ &\text{Decay of A:} && A_{i,j} \rightarrow \emptyset, \\ &\text{Division of A:} && B_{i,j} + A_{i,j} \rightarrow B_{i,j} + 2A_{i,j} \end{aligned} \tag{14}$$

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, four cells contain a population of 100 of species B, and the remainder of the cells contain no A or B respectively.

$$\begin{aligned} A_{i,j} &= \begin{cases} 100, & i = 10, j = 10, \\ 0, & \text{else} \end{cases} \\ B_{i,j} &= \begin{cases} 100, & \text{selected } i, j, \\ 0, & \text{else}. \end{cases} \end{aligned} \tag{15}$$

To use the shared memory efficiently, in addition to using the sparse matrix technique we group the M reac-

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.

tions 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 in each cell, 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* in each cell, and for species *B* we only store the cell position if the population of species *B* of that cell is not 0. 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 such as 100 × 100 in shared memory. To achieve good performance for those large models, the key is efficient use of the memory hierarchy. The basic idea is to block the most frequently used data on shared memory to minimize the global memory accesses. In our computation we store only the reaction rate for each type of reaction, the total population of *A*, the positions of the cells where the population of *B* is nonzero, and the population of *A* in all such cells with nonzero population of *B*, plus the cell where *A* is initialized. These data are potentially used more frequently during the computation than the population of *A* in other cells. With the above data in shared memory, we can calculate the next time step. To determine which reaction fires next, we first determine the reaction type of the next firing reaction. Then, to find the position of the next reaction to fire, we begin by searching the cells that are in the shared memory, as these are the most likely candidates. If we cannot find the next firing reaction position in the shared memory, we search the cells in the global memory. With the above method we can simulate this model on a large grid with excellent performance. (We also tried dynamic swapping data between shared memory and global memory according to the firing frequency, but the overhead of keeping track of firing frequency for each cell and data swapping between the shared memory and global memory actually slows down the performance.) For large problems, we cannot run too many realizations in parallel, 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. Often the SSA is used to generate ensembles (typically ten thousand to a million) of stochastic simulations. In this context, even the most efficient implementations of the SSA can be very time consuming. The current generation of GPUs appears to be very well-suited for this purpose. On the two model problems we tested, we observed speedups of 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 on the host workstation.

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

Acknowledgments

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.

Author Biographies

Dr Hong Li is a postdoctoral researcher in the department of Computer Science at the University of California, Santa Barbara. She received her B.S. in computer science from Peking University in 1998, and her M.A. and Ph.D. degrees in Computer Science from the University of California, Santa Barbara, in 2006 and 2008, respectively. Her research interests are multiscale algorithms and software for biochemical systems, and high performance stochastic simulation methods for chemically reacting systems, including sequential algorithms, parallelization, and use of advanced computer architectures (general-purpose GPU).

Dr Linda Petzold is currently Professor in the Department of Computer Science (Chair 2003–2007) and the Department of Mechanical Engineering, and Director of the Computational Science and Engineering Program at the University of California, Santa Barbara. She received her Ph.D. in Computer Science in 1978 from the University of Illinois. From 1978–1985 she was a member of the Applied Mathematics Group at Sandia National Laboratories in Livermore, California, from 1985–1991 she was Group Leader of the Numerical Mathematics Group at Lawrence Livermore National Laboratory, and from 1991–1997 she was Professor in the Department of Computer Science at the University of Minnesota. Dr. Petzold is a member of the US National Academy of Engineering. She is a Fellow of the ASME, SIAM and the AAAS. She was awarded the Wilkinson Prize for Numerical Software in 1991, the Dahlquist Prize in 1999, and the AWM/SIAM Sonia Kovalevski Prize in 2003.

Notes

1 A MAD is a multiply-add.

2 We note that the compiler-generated SSE code did not yield much improvement. This may be because of 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.

3 We note that this is still far from the theoretical peak GFLOPS on the GPU, which is not surprising given the number of non-floating point operations in discrete stochastic simulation.

References

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

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

Brent, R. P. (1992). *Uniform random number generators for vector and parallel computers*. In proceedings of 5th Australian Supercomputer Conference.

Cao, Y. and Petzold, L. (2006). Accuracy limitations and the measurement of errors in the stochastic simulation of chemically reacting systems, *J. Comput. Phys.*, **212**: 6–24.

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

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

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

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

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

Golub, G. H. and Loan, C. F. V. (1996). *Matrix computations*, 3rd edition, Baltimore: Johns Hopkins.

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

Li, H. and Petzold, L. (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

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

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

Li, H., Kolpas, A., Petzold, L., and Moehlis, J. (2009). Parallel simulation for a fish schooling model on a general-purpose graphics processing unit, *Concurrency and Computation: Practice and Experience*, **21**: 725–737.

Mascagni, M. (1999). SPRNG: A scalable library for pseudorandom number generation. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas.

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

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

Matsumoto, M. and Nishimura, T. (2000). Dynamic creation of pseudorandom number generators. In Monte Carlo and Quast – Monte Carlo Methods 1998 Edited by H. Niederreiter and J. Spanier. Berlin: Springer-Verlag, pp 56–69.

Matsumoto, M. and Nishimura, T. (2002). A nonempirical test on the weight of pseudorandom number generators. In *Monte*

Carlo and quasi-Monte Carlo methods 2000, edited by K. T. Fang, F. J. Hickernel, and H. Niederreiter, pp. 381–395. Berlin: Springer-Verlag.

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

McColluma, J. M., Peterson, G. D., Cox, C. D., Simpson, M. L., and Samatova, N. F. (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 Nadar, M. (2007). Stochastic DT-MRI connectivity mapping on the GPU, *IEEE Transactions on Visualization and Computer Graphics*, **13**: 1504–1511.

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

NVIDIA (2008b). NVIDIA CUDA Toolkit and SDK Release 1.0. http://news.developer.nvidia.com/2007/06/nvidia-cuda-too.html

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

Podlozhnyuk, V. (2008). Parallel Mersenne Twister. http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/MersenneTwister/doc/MersenneTwister.pdf

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

Shnerb, N., Louzoun, Y., Bettelheim, E., and Solomon, S. (2000). The importance of being discrete – life always wins on the surface, *Proc. Natl Acad. Sci. USA*, **97**: 10332.

Yoshimi, M., Osana, Y., lwaoka, Y., Funahashi, A., Hiroi, N., Shibata, Y., lwanaga, N., Kitano, H., and Amano, H. (2005). The design of scalable stochastic biochemical simulator on FPGA. *Proceeding of IEEE International Conference on Field Programmable Technologies (FPT2005)*, pp. 139–140.