GPU-Accelerated Stochastic Simulation of Biochemical Networks

SUMMARY We present a GPU (graphics processing unit) accelerated stochastic algorithm implementation for simulating biochemical reaction networks using the latest NVidia architecture. To effectively utilize the massive parallelism offered by the NVidia Pascal hardware, we apply a set of performance tuning methods and guidelines such as exploiting the architecture’s memory hierarchy in our algorithm implementation. Based on our experimentation results as well as comparative analysis using CPU-based implementations, we report our initial experiences on the performance of modern GPUs in the context of scientific computing.

key words: GPU computing, parallel programming, stochastic simulation

1. Introduction

Simulating biochemical reaction networks plays a major role in the understanding and the development of cell biology. In this area, incorporating stochasticity to biochemical network simulation, first pioneered by Gillespie [1], [2], has become crucial in securing analysis accuracy due to its ability to interpret natural stochastic phenomena in cell biology. Stochastic simulation of biochemical networks is usually computationally expensive because a large number of samples or realizations need to be generated to provide meaningful results. Therefore, a cluster of high-performance computers has traditionally been used for stochastic simulation.

For the last decade, however, many-core accelerators like GPUs have become a significant compute engine in the scientific computing area including stochastic simulation of biochemical systems, and there has been a sizable amount of research efforts on implementing stochastic simulation on the GPU. For instance, Li and Petzold [3] presented one of the first GPU-based implementations of Gillespie’s algorithm. Komarov and D’Souza [4] provide fine-grained parallelization of a single realization. However, these efforts are based on old versions of the GPU hardware, thus being unable to reflect upon recent advances in the fast-growing GPU technology.

In this paper, we present a GPU-based implementation of stochastic simulation algorithm and our work makes the following contributions:

- We implement biochemical stochastic simulation on the latest general purpose GPU architecture. To the best of our knowledge, there is no recent published work on GPU-accelerated simulations in this area and our work presents up-to-date stochastic simulation implementations and their performance results for modern GPU architectures.
- We apply and evaluate a set of performance optimization techniques for modern GPUs in the context of stochastic simulations. To achieve most optimized performance, we effectively make use of the massive hardware parallelism and analyze its performance results in terms of the Pascal architecture characteristics. In addition, we compare our GPU-based implementation over a CPU-based implementation on a cluster environment.

The remainder of this paper is organized as follows. Section 2 presents the background information for our work. Section 3 describes the details of our simulation algorithm implementation and its optimization methods. Section 4 provides experimental results of our approach and evaluates its effectiveness. Finally, we summarize our work and make conclusions in Sect. 5.

2. Background

In this section, we provide the background information by briefly describing the stochastic algorithm for simulating biochemical reaction networks and NVidia’s CUDA programming model for its GPU products.

2.1 Gillespie Stochastic Simulation Algorithm

Gillespie’s Stochastic Simulation Algorithm (SSA) [1], [2] is one of the favorite methods for simulating biochemical reaction networks and NVidia’s CUDA programming model for its GPU products.
time interval.

More specifically, the SSA can be summarized as follows. (1) Initialize the system with the time \( t = t_0 \) and the state \( x = x_0 \). (2) Calculate the propensity functions with the system in state \( x \) at time \( t \). (3) Generate random numbers for the firing time \( \tau \) of the next reaction and select the reaction \( R_j \) to fire. (4) Update the system state with the new population \( x + y_j \) at time \( t + \tau \). (5) Repeat from stage 2 unless the simulation has reached the final time.

2.2 NVidia CUDA Programming Model

A modern NVidia GPU hardware consists of a set of multi-processors, called Streaming Multiprocessors (SM), where each multiprocessor contains a number of “small” serial processor cores. Figure 1 shows a simplified view of the CUDA memory hierarchy. Here, a group of CUDA threads are organized into a thread block with shared memory accessible by all the threads of the block. In principle, a thread block can be configured as any of 1D, 2D, or 3D arrays of threads, and we show in the figure a 1D array of thread blocks because, as described later, we employ a 1D array of threads for our algorithm implementation. The thread blocks are distributed to the SMs, where threads are split into warps and all the threads in a warp execute synchronously on the SM.

Grouping a number of thread blocks defines a grid of blocks as shown in Fig. 1 and there are usually a number of grids which can be configured up to 3D like a thread block. In this work, we again use a 1D array of block grids for our implementation. In addition to shared memory of a thread block, a CUDA thread has a small number of registers and private local memory which is physically implemented in the off-chip DRAM of the GPU hardware. There is also off-chip global memory that can be accessed by all the threads. A piece of code to execute with the GPU is called a *kernel* in CUDA. A kernel is launched to execute with blocks of CUDA threads that are a multiple of 32 (warp size) and how many grids and threads are used is specified by the programmer using the *execution configuration* when the kernel is launched.

### Table 1 Specifications of GTX 1080 Ti and P100

| Specification                        | P100 | GTX 1080 Ti |
|-------------------------------------|------|------------|
| Microarchitecture                   | GP100| GP102      |
| Transistor Count                    | 15.3B| 12B        |
| SMs                                 | 56   | 28         |
| CUDA Cores/SM (FP32)                | 64   | 128        |
| Total CUDA Cores                    | 3584 | 3584       |
| Memory Size                         | 16GB | 11GB       |
| Memory Type                         | HBM2 | GDDR5X     |
| Memory Bandwidth                    | 7200GB/s | 484GB/s |
| Shared Memory/SM                    | 64KB | 64KB       |
| Register File/SM                    | 256KB| 256KB      |
| Max Warps/SM                        | 64   | 64         |
| FP32 Performance                    | 10.6TFLOPS | 11.3TFLOPS |
| FP64 Performance                    | 5.3TFLOPS | 0.355TFLOPS |

2.2.1 NVidia Pascal Architecture

Pascal [5] is the latest GPU architecture from NVidia, except that the next generation has been recently announced with the codename “Volta” [6] and its initial products are rumored being shipped to vendors like data centers from Aug. 2017. Volta-based graphics cards are reported to be released in the first quarter of 2018. Our work is based on GTX 1080 Ti, one of the latest Pascal-based graphics cards available in the market. GTX 1080 Ti has 28 SMs and 128 CUDA cores per SM, resulting in 11.3TFLOPS for single-precision (FP32) with its 3584 cores. Compared to P100, a more compute-oriented hardware based on Pascal, 1080 Ti focuses more heavily on the single-precision (FP32) performance but lacks in the double-precision performance (FP64). In addition, 1080 Ti shows lower memory bandwidth due to using GDDR5X instead of HBM2 (second generation of high bandwidth memory). Stochastic simulation algorithms for biochemical networks do not require double-precision in general, and our implementation in this work uses single-precision calculations, which allows for utilizing 1080 Ti’s performance characteristics. 1080 Ti’s specification is shown in Table 1 compared with P100.

### 3. SSA Algorithm Implementation on NVidia Pascal

3.1 Target Network: Fast Reversible Isomerization

To readily observe the performance potentials of the latest NVidia GPU hardware, we consider a simple biochemical network for the simulation called “fast reversible isomerization” with “fast” and “slow” reactions mixed together [7] in the network. It can be described as the following equations,

\[
S_1 \xrightleftharpoons[c_1]{c_2} S_2 \quad \text{(fast)}, \quad S_2 \xrightarrow{c_3} S_3 \quad \text{(slow)},
\]

where \( S_i \) denotes the species and \( c_j \) denotes the reaction rate constant. With \( x_i(t) \) denoting the number of species \( S_i \) at time \( t \), the propensity functions and state-changing vectors can be specified as follows,

\[
a_1(x) = c_1 x_1, \quad v_1 = (-1, +1, 0),
\]

\[
a_2(x) = c_2 x_2, \quad v_2 = (+1, -1, 0),
\]

\[
a_3(x) = c_3 x_2, \quad v_3 = (0, 0, +1),
\]
\[ a_2(x) = c_2 x_2, \quad \nu_2 = (+1, -1, 0), \]
\[ a_3(x) = c_3 x_3, \quad \nu_3 = (0, -1, +1). \]

For our measurement and evaluation purposes with the fast reversible isomerization simulations, we specifically use the following parameter values.

\[ c_1 = 1, \quad c_2 = 2, \quad c_3 = 5 \times 10^{-5}, \]
\[ x_1 = 1200, \quad x_2 = 600, \quad x_3 = 0 \quad \text{at} \quad t = 0, \]
\[ \text{FINALTIME} = 1000, \]

where FINALTIME denotes the simulation finish time.

### 3.2 Implementation and Optimization

Programming the NVidia GPUs and their optimization approaches are well documented in NVidia’s CUDA programming guides [8], [9]. In addition, architecture-specific optimization opportunities separately provided for different microarchitectures. Since we target the Pascal architecture, the Pascal tuning documentation [10] is relevant here.

The first step for utilizing parallel architectures with any given simulation algorithm is to identify parallel optimization points in sequential code. The Gillespie SSA algorithm itself opens the door for massive parallelism since multiple reaction realizations can be simulated in parallel with virtually negligible communication among realizations: hence, the algorithm can be categorized as *embarrassingly parallel*.

Our scheme for implementing the Gillespie algorithm takes the following steps in the code.

- Allocate global memory in the GPU device for storing random numbers.
- Execute a GPU kernel for random number initialization.
- Allocate GPU memory for storing species in the biochemical network.
- Copy the initial values of the species from the host to the GPU.
- Launch the Gillespie SSA algorithm kernel.
- Copy the final values of the species from the GPU to the host

### Shared Memory on GPU

Minimizing the access to the GPU global memory inside the GPU SSA kernel is one of the key optimization methods in general since it takes orders-of-magnitude longer time than accessing other memory in the GPU such as registers or shared memory. Our implementation effectively makes use of shared memory to reduce global memory accesses, thereby improving the overall performance. By using shared memory, we can also reduce any performance slowdown that can be caused by misaligned accesses to the global memory.

In the SSA kernel implementation, we put and manage most of the data in shared memory which has roughly 100\% lower latency. The data we manage in shared memory include each population of the species, state-changing vectors, and reaction rate constants. While state-changing vectors and reaction rate constants require very small storage and are shared by all the threads in a block, the population information is different among different evolutions that are separately simulated by individual threads, which increasingly consumes shared memory as the number of samples increases. For our fast reversible isomerization case where the number of species is 3, the total number of the species data to manage in shared memory per block becomes \(3 \times \text{number of threads per block}\), which should not be bigger than the total amount of shared memory per block, 49152 bytes in our case for GTX 1080 Ti, to maintain improved performance.

### Kernel Execution Configuration

It is important to keep the GPU hardware efficiently utilized during the simulation. In CUDA programming, the SM processor occupancy is one of the major metrics to illustrate how well the SMs are occupied by computations such that memory access latencies are effectively hidden for improved application performance. The kernel execution configuration is the key factor for occupancy and the occupancy value can be calculated by entering kernel code information including the thread block size and register/shared memory usage into the “occupancy calculator” [11] offered by NVidia.

Our SSA kernel implementation uses 26 registers and 560 bytes of shared memory for a 1D thread block with 32 threads. Figure 2 shows occupancy values with varying thread block sizes, where only 32 warps are occupying each SM which support 64 active warps (Table 1), resulting in 50\% occupancy in our case. As indicated in the figure, we may reach 100\% occupancy by increasing the thread block size to 64. However it turns out that 100\% occupancy with 64 threads actually slow down the performance. For instance, using 64 threads for simulating 128K samples from the fast reversible isomerization process took 95 seconds while using 32 threads for the same number of samples took only 54 seconds. This is mainly because increased threads in a block happen to require more registers that can otherwise be effectively used to improve thread performance [12].

![Impact of Varying Block Size](image-url)
Pascal-based GPUs with the CUDA 8.0 support are equipped with a diverse set of new features including atomics for double-precision operations, unified memory, and the C++ lambda expression support. However, the SSA simulations do not require double-precision atomics, and unified memory management and the lambda expression support are rather programming productivity related features which are not directly affect the performance. Therefore, we do not consider Pascal-specific optimization opportunities any further in this work.

4. Performance Evaluation

Using our GPU SSA implementation, we simulated the fast reversible isomerization process on a Linux 64bit machine (Ubuntu 16.04 LTS) with Intel Core i7-7700 @ 3.60Ghz, 64GB DDR4 memory, and a 512GB NVMe SSD. To evaluate performance scaling as the problem size (i.e., number of samples) grows, we varied the grid size (i.e., number of thread blocks) from 256 (2³), to 4096 (2¹²), and to 32769 (2¹⁵) while the thread block size is fixed to 32. Hence, the number of samples in each simulation is 2¹³, 2¹⁷, and 2²⁰. To compare our GPU implementation with a CPU implementation, we used the StochKit [13] implementation on a Linux cluster where each compute node comprises 2x Intel Xeon E5-2470v2 10 core @ 2.40Ghz (i.e., 20 cores and 40 hardware threads in total), 96GB Main memory @ 1600MT/s, and 80GB SSD system drive, interconnected with 10Gbit Ethernet. Specifically, we used 256 processors on 16 nodes on the cluster for simulation. Table 2 shows statistical properties of our GPU implementation for simulating 2¹³ samples, which are virtually identical to those of the cluster version.

Figure 3 shows the plots of execution time over the number of simulated samples using our GPU implementation and StochKit. While both implementations show linear scaling with the problem size, our GPU implementation consistently shows about 3× better performance than StochKit, thus indicating that GPUs are very well suited to stochastic simulations and latest versions of the GPU hardware can be more powerful than CPU-based clusters in this kind of scientific computing settings.

5. Conclusions and Future Work

We presented a GPU-based stochastic algorithm implementation for simulating biochemical networks using one of the latest NVidia GPU architectures, in which we applied a set of optimization techniques offered by the CUDA programming model. Our implementation with a few typical optimizations on a single GPU device delivers 3× faster performance than a CPU-based implementation on a 16-node cluster environment, which shows that GPUs can be very powerful for some scientific computing applications if the used algorithm can be effectively parallelized following an adequate GPU programming model.

Based on the work in this paper, we plan to evaluate latest GPU architectures further in accelerating stochastic simulations by extending our implementation to consider larger and more complex biochemical networks where the architectural characteristics of the GPU such as the memory hierarchy and kernel execution configurations need to be examined in depth and properly adjusted.

Acknowledgements

This work was supported by a 2016 research grant from Youngsan University, Republic of Korea. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP; Ministry of Science) (No. 2017R1C1B2009361).

References

[1] D.T. Gillespie, “A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Reactions,” J. Comput. Phys, vol.22, no.4, pp.403–434, Dec. 1976.
[2] D.T. Gillespie, “Exact Stochastic Simulation of Coupled Chemical Reactions,” The Journal of Physical Chemistry, vol.81, no.25, pp.2340–2361, 1977.
[3] H. Li and L. Petzold, “Efficient Parallelization of the Stochastic Simulation Algorithm for Chemically Reacting Systems on the Graphics Processing Unit,” The International Journal of High Performance Computing Applications, vol.24, no.2, pp.107–116, 2010.
[4] I. Komarov, R.M. D’Souza, and M. Schilstra, “Accelerating the Gillespie Exact Stochastic Simulation Algorithm using Hybrid Parallel Execution on Graphics Processing Units,” PLoS One, vol.7, no.11, p.e46693, 2012.
[5] NVidia, “NVidia Tesla P100 Whitepaper,” 2016. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.
[6] NVidia, “NVidia Tesla V100 GPU Architecture v1.0,” 2017. http://www.nvidia.com/content/gated-pdfs/Volta-Architecture-Whitepaper-v1.0.pdf.
[7] Y. Cao, D.T. Gillespie, and L.R. Petzold, “The Slow-Scale Stochastic Simulation Algorithm,” J. Chem. Phys, vol.122, no.1, p.014116, Jan. 2005.
[8] NVidia, “CUDA C Programming Guide 8.0,” 2017. http://docs.
nvidia.com/cuda/cuda-c-programming-guide.

[9] NVidia, “CUDA C Best Practices Guide 8.0,” 2017. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide.

[10] NVidia, “Pascal Tuning Guide 8.0,” 2017. http://docs.nvidia.com/cuda/pascal-tuning-guide.

[11] NVidia, “CUDA Occupancy Calculator,” 2007. http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls.

[12] V. Volkov, “Better Performance at Lower Occupancy,” Proceedings of the GPU Technology Conference, 2010.

[13] H. Li, Y. Cao, L.R. Petzold, and D.T. Gillespie, “Algorithms and Software for Stochastic Simulation of Biochemical Reacting Systems,” Biotechnology Progress, vol.24, no.1, pp.56–61, 2008.