Accelerating Stochastic Simulations on GPUs Using OpenCL

Pilsung KANG\(^{1(a)}\), Member

SUMMARY Since first introduced in 2008 with the 1.0 specification, OpenCL has steadily evolved over the decade to increase its support for heterogeneous parallel systems. In this paper, we accelerate stochastic simulation of biochemical reaction networks on modern GPUs (graphics processing units) by means of the OpenCL programming language. In implementing the OpenCL version of the stochastic simulation algorithm, we carefully apply its data-parallel execution model to optimize the performance provided by the underlying hardware parallelism of the modern GPUs. To evaluate our OpenCL implementation of the stochastic simulation algorithm, we perform a comparative analysis in terms of the performance using the CPU-based cluster implementation and the NVidia CUDA implementation. In addition to the initial report on the performance of OpenCL on GPUs, we also discuss applicability and programmability of OpenCL in the context of GPU-based scientific computing.

key words: GPU computing, OpenCL, parallel programming, stochastic simulation

1. Introduction

The study of biochemical reaction networks is fundamental to the development of many different biology areas such as cell biology and drug science. Traditionally, simulating biochemical reaction networks has been performed using two major approaches. The first one is deterministic approaches that model the reaction network using ordinary differential equations (ODEs). Although these approaches are mathematically intact, they can be inaccurate for systems with small populations because they fail to consider statistical fluctuations innate to natural phenomena.

The second major approach is based on Gillespie’s stochastic simulation algorithm (GSSA)\([1], [2]\), which overcomes the issues of the deterministic approaches by adopting stochastic methods. The GSSA-based methods have typically been very time-consuming because they need to produce a very large number of realizations to obtain statistically accurate description of the simulated system.

In this paper, we present an OpenCL\([3]\) implementation of the GSSA and investigate the issues in parallelizing the algorithm on modern GPUs. Specifically, our work makes the following contributions:

- We implement an OpenCL-based acceleration of the GSSA on one of the latest NVidia GPU hardware. Although there has been a sizable amount of work on implementing GPU acceleration for the GSSA\([4], [5]\), they are all based on the NVidia CUDA programming model. In contrast, this paper is the first report on the GSSA using OpenCL to the best of our knowledge.
- By comparing our OpenCL-based GSSA implementation with other implementations in other major heterogeneous programming frameworks including NVidia CUDA and OpenACC, we present an initial comparative evaluation for different programming models on the modern GPU architecture. While there has been extensive research on this subject such as\([6]\), our work is specific to the computational biology domain and can be useful in programming GPU-based heterogeneous systems in the scientific computing community.

We note that this paper extends our previous work\([7]\) where we used the NVidia CUDA framework for implementing the stochastic simulation algorithm on the GPU. In contrast, this paper investigates the use of the OpenCL programming model for the GPU parallelization of the stochastic simulation. The remainder of this paper is organized as follows. Section 2 sets the context by presenting the background information for the work in the paper. Section 3 describes our OpenCL implementation of the stochastic simulation algorithm in detail. Section 4 evaluates our OpenCL implementation. Finally, we summarize our work and make conclusions in Sect. 5.

2. Background

In this section, we briefly discuss the stochastic simulation algorithm for simulating biochemical reaction networks and provide a concise description of the OpenCL model for programming GPUs.

2.1 Stochastic Simulation Algorithm

The stochastic simulation algorithm was first introduced by Gillespie for evolving biochemical systems of reactions\([1], [2]\). Although a concise description of the GSSA can be found in our previous work\([7]\), we briefly summarize the GSSA again in the following.

Consider a biochemical reaction system where there are \(N\) molecular species which we call \(S_1, \ldots, S_N\), respectively. For the population or the number of each species at time \(t\), we use \(X(t) = (X_1(t), \ldots, X_N(t))\). In the reaction system, the species chemically interact with each other, which
can be described with reaction channels, \( R_1, \ldots, R_M \). Each channel \( R_j \) is expressed with two components: the state-changing vector and the propensity function. First, the state-changing vector, denoted as \( v_j = (v_{j1}, \ldots, v_{jN}) \), defines the amount of change in the number of species \( S_i \) when the reaction \( R_j \) occurs. The propensity function, denoted as \( a_i \), is used to define the probability that the \( R_j \) reaction will occur in the next time interval. Specifically, the probability that one \( R_j \) reaction will occur in \([t, t + dt]\) can be written as \( a_i(x)dt \). The GSSA evolves the reaction system at each step by randomly choosing a reaction channel to fire and the amount of time interval to increase. With this setup, the GSSA can be summarized as follows:

1. **Initialization**: Initialize the system with the time \( t = t_0 \) and the state \( x = x_0 \).
2. **Propensity calculation**: Calculate the propensity functions \( a_i \) for the system in state \( x \) at time \( t \).
3. **Reaction channel selection**: Based on the propensity calculation, randomly select the reaction \( R_j \) to fire and calculate the firing time \( \tau \) of \( R_j \).
4. **System update**: Change the population of each species by \( x \leftarrow x + v_j \) and update the system time by \( \tau \).
5. **Termination**: Go back to step 2 if the simulation has not reached the final time. Otherwise, stop the simulation.

Although the GSSA had been very computationally expensive to perform, the recent advent of modern GPUs with a great amount of compute powers is increasingly allowing for efficient execution of the GSSA. In this regard, this paper studies one of the modern GPU programming models, OpenCL, in the context of implementing the GSSA.

### 2.2 OpenCL

OpenCL™ (Open Computing Language) is an open standard for programming heterogeneous parallel systems. The foremost characteristic of OpenCL is functional portability, which allows a single source to be used across different hardware and architectures.

In OpenCL, the system consists of one host with at least one compute device. A compute device, in turn, consists of one or more compute units, and a compute unit is further divided into one or more processing elements (PEs). For an NVidia GPU card as an example, the card itself is considered as a compute device, the streaming multiprocessors (SMs) as compute units, and the CUDA cores as PEs in OpenCL, respectively. The basic unit of work on a device is called a work-item and a work-item is mapped to a PE in the device. Work-items are grouped to form a workgroup, whose size and dimensionality is specified by the programmer. The code specifying a work-item to execute on a PE is called a kernel. Hence, when a kernel is launched on a compute device, the kernel code is executed in parallel by all the PEs in the workgroup. Later in Sect. 3, we describe our OpenCL kernel implementing the GSSA.

Figure 1 shows a logical view of the memory hierar-

![OpenCL memory model](image)

**Fig. 1** OpenCL memory model [8]

### 3. OpenCL Implementation of GSSA on GPU

#### 3.1 Simulation Problem: Fast Reversible Isomerization

As an input to the GSSA simulation target, we choose the fast reversible isomerization problem [9], as we did in our previous works [7], [10]. This is to compare our OpenCL implementation with the CUDA and the OpenACC implementations in the same settings.

The fast reversible isomerization problem we consider is a simple reaction process with only 3 species, where 2 species change their population quite “fast” and the population of the remaining species is not very affected by the fast reactions. A detailed information and the settings on this reaction process can be found in our previous work [7].

#### 3.2 OpenCL Implementation of GSSA

Figure 2 shows the major part of the GSSA function, ssa_kernel, implemented as an OpenCL kernel. ssa_kernel takes the location of the species population in the GPU global memory as input, specified as \( \_\text{global} \ \text{int}^* \ x \) at line 4, along with the seed value for the random number generation. The input array of the species pointed by \( x \) holds the globalsize copies of the initial population values of the species in the reaction network. For the fast reversible isomerization process as an example, there are 3 species (12 bytes since \( \text{int} \) takes 4 bytes) in the system and if we simulate 1024 samples or instances of the system, then the input

```c
... // Kernel code implementing the GSSA...```

...
array will have 1024 copies of the 3 species, resulting in 12288 bytes in total.

Since the access to the local memory is typically an order of magnitude faster than to the global memory in GPUs, we use a local memory to store the species population for faster memory access (line 10) and copy the initial values from the global memory (line 16). In doing so, we map between the indices of the global memory (line 13) and the local memory (line 14) by utilizing the global id (line 7) and the local id (line 8), in which $nx$ is the number of species in the system and $xblocksize$ is the size of the local group.

For the random number generator (RNG), we use Tiny Mersenne Twister (TinyMT)\[12\] v1.1 (line 22-24) as it is readily available for OpenCL programming and very easy to use. In fact, unlike the NVidia CUDA framework that includes an RNG library called cuRAND \[13\], there is no standard or popular RNG library upfront, whereas there are only a few RNG libraries publically available \[14\], \[15\]. This makes OpenCL programming for stochastic simulations somewhat inconvenient because the selection and use of the RNG library is quintessential for stochastic simulations. TinyMT allows to generate pseudo-random numbers whose periods are $2^{127} - 1$, while CUDA cuRAND generates pseudo-random numbers with a period greater than $2^{190}$.

The while loop at line 26 is a straightforward implementation of the GSSA. First, we calculate propensity values for each reaction channel and accordingly choose the channel to fire in the next step (line 29-32). Then, we fire the chosen channel and update each species’ population using the state-changing vector information (line 34-35). Finally, we calculate the time step to advance in the simulation and update the current simulated time (line 37), followed by preparing new random numbers to use in the next iteration of the loop (line 39-40).

4. Performance Evaluation

To evaluate our OpenCL implementation of the GSSA, we use GTX 1080 Ti, one of the recent versions of the NVidia GPUs. GTX 1080 Ti offers 28 SMs or compute units in the OpenCL terminology. Each SM contains 128 CUDA cores or PEs, resulting in 3585 cores in total. The host is a 64-bit Linux machine with Ubuntu 16.04 LTS equipped with Intel Core i7-7700 @ 3.60Ghz, 64GB DDR4 memory, and a 512GB NVMe SSD. The used compiler is GCC 5.4.0. We also use the NVidia driver v384.130 which implements version 1.2 of the OpenCL specification.

Table 1 shows statistical properties of the fast isomerization problem simulated for $2^{13}$ samples by our OpenCL implementation. For comparison, the table also shows the simulation results from other implementations using OpenACC and NVidia CUDA on the same GPU. The table also shows the simulation results of the CPU-based implementation using StochKit\[16\] on a cluster. We note that these simulation results were previously reported in our works\[7\], \[10\] and borrowed here for reference. In the table, our OpenCL implementation shows essentially the same simulation behavior as the other versions.

Regarding the performance aspect, Fig. 3 compares the execution time of the different versions of the GSSA for varied number of simulation samples. As previously reported, the OpenACC implementation performs slightly better than the CUDA implementation, while these GPU versions are about 3× faster compared to the cluster version. Interestingly, the OpenCL version shows 2.9× to 3.4× better performance that the other GPU versions for $2^{17}$ and $2^{10}$ samples. The performance improvement is mainly due to the TinyMT RNG in our OpenCL GSSA implementation. In general, the performance behavior of stochastic simulation relies heavily on the used RNG \[17\]. As previously described, TinyMT’s period of the generated random number sequence is $2^{127} - 1$, which is far shorter than $2^{190}$, the lower bound for the period.

| $X_1$ (fast) | Mean | Std  | $X_2$ (fast) | Mean | Std  | $X_1$ (slow) | Mean | Std  |
|-------------|------|------|-------------|------|------|-------------|------|------|
| Cuda (GPU)  | 1180.51 | 20.41 | 589.93 | 20.18 | 29.56 | 5.37 |
| OpenACC (GPU) | 1180.28 | 20.15 | 590.34 | 19.25 | 29.41 | 5.43 |
| StochKit (cluster) | 1180.03 | 20.29 | 590.32 | 20.11 | 29.65 | 5.42 |
of the cuRAND library commonly used in the CUDA and the OpenACC implementations. Therefore, without having to guarantee as long a period as cuRAND, TinyMT can operate much faster accordingly. Hence, we note that the performance numbers shown here for different GPU programming models are not something to directly compare with one-on-one basis due to their different settings.

Besides, we additionally tested our OpenCL implementation of the GSSA on the same Intel i7-7700 Linux machine, but this time targeting the Intel CPU instead of the NVidia GPU device. This was to ensure that our OpenCL implementation was functionally correct even when tested against a different architecture (i.e., the CPU). In addition, we intended to confirm the portability advantage of using OpenCL. Indeed, after setting up Intel’s OpenCL runtime 16.1.1 on the machine, the modification needed in our implementation was only at the call to clGetDeviceIDs, where its second argument has been changed from CL_DEVICE_TYPE_GPU to CL_DEVICE_TYPE_CPU. While the performance of the CPU version was about 50× to 90× slower than the GPU version, the performance behavior between the two are not directly comparable due to architectural differences and capabilities.

5. Conclusions and Future Work

We presented an OpenCL implementation of the GSSA for simulating biochemical reaction systems on GPUs. Our GSSA kernel exploits the memory hierarchy and level of parallelism of the GPU hardware so that most of the memory accesses in the kernel occur around the faster memory like local and private memory, while avoiding frequent accesses to the slow global memory. Our OpenCL implementation shows better performance than other GPU implementations, but this is largely due to the use of a faster but rather “low-quality” RNG. In the future, we plan to investigate a more thorough and systematic way of profiling against our OpenCL implementation as well as analyzing performance measurements in depth.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2017R1C1B2009361).

References

[1] D.T. Gillespie, “A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Reactions,” J. Computational Physics, 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] J.E. Stone, D. Gohara, and G. Shi, “OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems,” Computing in Science Engineering, vol.12, no.3, pp.66–73, May 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] K. Sumiyoishi, K. Hirata, N. Hiroi, and A. Funahashi, “Acceleration of Discrete Stochastic Biochemical Simulation using GPGPU,” Frontiers in Physiology, vol.6, p.42, 2015.

[6] S. Memeti, L. Li, S. Pilana, J. Kołodziej, and C. Kessler, “Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption,” ARMS-CC ’17, pp.1–6, ACM, 2017.

[7] P. Kang, “GPU-accelerated Stochastic Simulation of Biochemical Networks,” IEICE Trans. Information and Systems, vol.E101-D, no.3, pp.786–790, 2018.

[8] Khronos OpenCL Working Group, “The OpenCL Specification v2.2,” 2018, https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf.

[9] Y. Cao, D.T. Gillespie, and L.R. Petzold, “The Slow-Scale Stochastic Simulation Algorithm,” Journal of Chemical Physics, vol.122, no.1, p.014116, Jan. 2005.

[10] P. Kang, “OpenACC Parallelization of Stochastic Simulations on GPUs,” IEICE Trans. Inf. & Syst., vol.E102-D, no.8, pp.1565–1568, 2019.

[11] “OpenACC,” http://www.openacc-standard.org.

[12] M.M.M. Saito, “Tiny Mersenne Twister (TinyMT): A small-sized variant of Mersenne Twister,” 2011, http://www.math.sci.hiroshima-u.ac.jp/~m-mat/TINYMT/index.html [accessed on Feb. 7, 2019].

[13] NVidia, “cuRAND,” https://docs.nvidia.com/cuda/curand/index.html.

[14] D.B. Thomas, “The MWC64X Random Number Generator,” 2011, http://cas.ee.ic.ac.uk/people/dt10/research/rngs-gpu-mwc64x.html [accessed on Feb. 7, 2019].

[15] P. L’Ecuyer, D. Munger, and N. Kemercho, “cRNGL: A Random Number API with Multiple Streams for OpenCL,” 2015, http://www.iro.umontreal.ca/~lecuyer/myftp/papers/clrng-api.pdf [accessed on Sept. 12, 2019].

[16] H. Li, Y. Cao, L.R. Petzold, and D.T. Gillespie, “Algorithms and Software for Stochastic Simulation of Biochemical Reactions Systems,” Biotechnology Progress, vol.24, pp.56–61, 2008.

[17] J. Passerat-Palmbach, C. Mazel, and D.R.C. Hill, “Pseudo-random streams for distributed and parallel stochastic simulations on gp-gpu,” Journal of Simulation, vol.6, no.3, pp.141–151, Aug. 2012.