OpenACC Parallelization of Stochastic Simulations on GPUs

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

SUMMARY  We present an OpenACC-based parallelization implementation of stochastic algorithms for simulating biochemical reaction networks on modern GPUs (graphics processing units). To investigate the effectiveness of using OpenACC for leveraging the massive hardware parallelism of the GPU architecture, we carefully apply OpenACC’s language constructs and mechanisms to implementing a parallel version of stochastic simulation algorithms on the GPU. Using our OpenACC implementation in comparison to both the NVIDIA CUDA and the CPU-based implementations, we report our initial experiences on OpenACC’s performance and programming productivity in the context of GPU-accelerated scientific computing.

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

1. Introduction

Since first pioneered by Gillespie\([1, 2]\) in 1970s, the stochastic method for simulating biochemical networks has become one of the major techniques for understanding cell biology and exploring its evolutionary process. However, stochastic simulation in scientific computing has been traditionally expensive due to its innate operational mechanism where a large amount of simulation samples are required for any significant analytical results. Therefore, a cluster of high-performant parallel systems or similar settings were typically used in performing stochastic simulation of biochemical networks. This trend is now changing with increasingly growing use of modern graphics hardware consisting of numerous small processors that execute in parallel.

In this paper, we present an OpenACC\([3]\) parallelization implementation of Gillespie’s stochastic simulation algorithm (GSSA) on modern GPUs and our work makes the following contributions:

- We implement an OpenACC-based GSSA for simulating biochemical reaction networks on the modern GPU architecture. To the best of our knowledge, there is no recent published work on OpenACC parallelization for GPUs in this area and our work presents the first report on the OpenACC implementation of the GSSA and its performance results for modern GPUs.
- Based on our experience, we evaluate the performance of the OpenACC implementation on the GPU and provides a comparative analysis against an NVIDIA CUDA\([4]\) implementation and a CPU-based implementation on a cluster environment in the context of stochastic simulations. Although there is a number of comparative evaluations between CUDA and OpenACC\([5, 6]\), our work is based on a different scientific computing domain and can provide distinct perspectives on the performance of OpenACC parallelization on the GPU for simulating biochemical reaction networks.

Our previous work implemented a CUDA version of the stochastic simulation algorithm\([7]\) and as an extension, we investigate OpenACC parallelization of the stochastic simulation in this paper. The remainder of this paper is organized as follows. Section 2 provides the background information for the discussions of the paper. Section 3 describes the details of our OpenACC implementation of the GSSA. Section 4 provides experimental results and evaluations of our approach. Finally, we summarize our work and make conclusions in Sect. 5.

2. Background

To set the context for our work in this paper, we present a brief description of the stochastic computational algorithm for simulating biochemical networks and the OpenACC parallel programming model.

2.1 Stochastic Simulation Algorithm

The GSSA models a well-stirred chemically reacting system of N molecular species \(S_1, \ldots, S_N\) represented by the dynamical state vector \(X(t) = (X_1(t), \ldots, X_N(t))\) where \(X_i(t)\) denotes the number of species \(S_i\) at time \(t\). It also considers the species interacting through \(M\) chemical reaction channels \(R_1, \ldots, R_M\), where each channel \(R_j\) is described by the state-changing vector \(v_j = (v_{1j}, \ldots, v_{Nj})\) and a propensity function \(a_j: v_{ij}\) is defined to be the change in the number of species \(S_i\) caused by one \(R_j\) reaction and \(a_j(x)dt\) is the probability that one \(R_j\) reaction event will occur in the next infinitesimal time interval \([t, t + dt]\). The GSSA evolves the system by generating random samples for \(j\) and \(\tau\), so that reaction \(R_j\) will be fired at \(t + \tau\). For the detailed description and operations of the SSA, we refer the readers to \([1, 2]\).

Because the GSSA method requires a significant number of simulation samples for drawing statistically meaningful results while it needs to simulate every event specified by...
the reaction channels, the GSSA had been traditionally considered very computationally expensive and practically difficult in the practical sense. However, the advent and adoption of the modern, powerful accelerators like GPU that offer a massive amount of compute cores enabled both an effective and affordable parallelization of the GSSA simulation. Following this recent trends in scientific computing, our work in this paper parallelizes the GSSA on the latest GPU architecture using OpenACC.

2.2 OpenACC

OpenACC is a directive-based programming model for parallel systems, which can be used for mainly heterogeneous (i.e., host CPU with attached accelerators) parallel systems. Similar to OpenMP [8], OpenACC makes use of a rich set of compiler directives to annotate compute-intensive regions that the programmer wants to parallelize and execute on the accelerator, whereas other parts of code are executed by host CPUs. Specifically, the compiler directive in OpenACC starts with `#pragma acc` and the compiler is generally free to ignore the directive when it does not support the directive or when it decides not to translate the directive. In addition to compiler directives, OpenACC defines a set of high-level runtime APIs (application programming interfaces) to help specify program execution configurations [9].

Figure 1 illustrates how a serial code can be parallelized for accelerator-based systems using OpenACC constructs. Here the SAXPY [10] for loop is annotated by the `#pragma directive, which directs the compiler to define a parallel computation `kernel` for the loop and offload the kernel onto the accelerator consisting of a great number of small compute cores. The rest of the code is executed by the CPU.

Similar to the CUDA programming model where different levels of parallelism are present such as grids and thread blocks, OpenACC also allows for a hierarchy of parallelism levels in the means of `gangs, workers, and vectors`. We note that OpenACC is a highly portable language targeting a wide range of different accelerators and in the case of GPU, there is no direct mapping to CUDA threads or blocks. It is up to the compiler regarding how to map OpenACC parallelism to the underlying CUDA hardware parallelism.

However, in a conceptual sense, OpenACC gangs can be mapped to the CUDA thread blocks, OpenACC workers to the CUDA warps, and OpenACC vectors to the CUDA SIMT (single instruction, multiple thread) threads [11].

3. OpenACC Parallelization of GSSA on NVidia Pascal

3.1 Simulation Problem: Fast Reversible Isomerization

Similar to our previous work [7], we choose the fast reversible isomerization problem [12] as our example biochemical network in order to compare the OpenACC implementation with the NVidia CUDA implementation in the same settings. Our fast reversible isomerization problem consists of both “fast” and “slow” reactions combined together and can be described by the following equations,

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

where \( S_i \) and \( c_j \) denote the species and the reaction rate constant, respectively. Further details of the fast reversible isomerization process and its simulation parameters can be found in our previous work [7].

3.2 OpenACC Implementation

The core of parallelizing the GSSA using OpenACC lies in the part where the massive amount of samples are evolved stochastically following the reaction rules among the species. These evolutions are independent from each other resulting in natural data-independence, which makes the simulation suitable for parallel execution. Considering this parallelization point, we implement the OpenACC version of the GSSA as in Fig. 2 that shows major elements of the algorithm.

Data Movement between Host and GPU

The `ssa_kernel` function takes the input array `x` which holds the initial population of each species comprising the reaction system. The size of the array is the number of species multiplied by the number of samples, `ns`, to simulate (line 1). Unlike the NVidia CUDA programming model where every data movement between the host and the device needs to be specified explicitly by the programmer, it can sometimes be implicitly performed by the compiler in OpenACC when the context is obvious, which makes programming in OpenACC easier and more productive. In our OpenACC implementation, we use this capability of OpenACC for data movement. That is, the species population data pointed by `x` is first allocated at the host side in the beginning, and the data is copied to the GPU memory later via data movement operations implicitly generated by the compiler when the core of the GSSA computation is offloaded to the GPU for parallel execution.

In the same manner, when the parallel computation is done at the GPU side, the species population data updated at the GPU side is transparently copied back to the
host side. All of these data movement operations are automatically generated by the OpenACC compiler for execution at both the beginning and the end of the GPU code, whereas we had to explicitly specify the necessary operations in our CUDA implementation of the GSSA [7]. Besides the compiler-generated implicit data movement operations, OpenACC also offers the data directive and a set of accompanying clauses to specify different data movements explicitly when needed [9].

Kernel Launch Configuration

After we prepare the seed and states of random number generation where we use the NVidia cuRAND library [13] (line 5), we define a parallel region to execute on the GPU using 256 thread blocks (num_gangs) and 256 threads in a block (vector_length) (line 8-9). In fact, specifying these kernel launch configuration parameters can undermine code portability, which is one of the most prominent features of the OpenACC framework, because the size and the level of parallelism are different across hardware architectures. Once explicitly specified in the code for some execution environment, num_gangs and vector_length need to be modified for optimal performance when the code is reused for another execution environment with different accelerators.

In this work, however, in order to fully maximize the performance of our GPU hardware, GTX 1080 Ti in our experiments, we fine-tune these OpenACC parameters at the expense of portability. Specifically, we use 256 gangs (thread blocks in CUDA) and 256 vectors (threads in CUDA) for launching the GSSA kernel. This is in contrast with the CUDA GSSA implementation in our previous work where 256 blocks with 32 threads show the best performance – increasing the number of threads in a block slowed down the performance on the contrary due to increased allocation and overhead of the registers in the GPU hardware. The OpenACC compiler used here makes careful use of the registers in generating the GPU code, thus avoiding negative effects on the performance with increased threads in a block.

At line 13, the outermost for loop is parallelized at both the gang and the vector levels, and the loop iterations are specified to execute independently among the threads, which utilizes the aforementioned intrinsic property of the stochastic simulation.

Accelerator Memory Hierarchy and Data Placement

The effective use of the CUDA shared memory in the NVidia GPUs is one of the key factors for optimizing application performance. While the memory hierarchy is explicit and visible to the programmer in CUDA, OpenACC’s memory model is very abstract to provide only one level of device memory. Instead, it is up to the OpenACC compiler to exploit the given accelerator’s memory hierarchy to arrange adequate data placement. Therefore, we do not perform any arrangement regarding data placement in our OpenACC implementation, which is in contrast with our CUDA implementation where we placed the species population data on the CUDA shared memory for faster access [7]. However, via NVidia’s profiling tool (nvprof), we confirmed that the CUDA shared memory is fully utilized at our OpenACC implementation’s runtime.

The rest of the code is a straightforward implementation of the GSSA. At line 17 and below, we initialize the computation states such as species populations, reaction channels, and random number generation. The while loop at line 22 iteratively evolves the reaction network until the simulation finish time by choosing the channel to fire at the next step (line 24), firing the chosen channel (line 29), and updating the system time (line 31 and below). The #pragma acc routine(logf) seq directive at line 31 tells the compiler to generate accelerator code for the logf function that will run sequentially in one thread on the accelerator.

4. Performance Evaluation

We evaluate our OpenACC implementation of the GSSA on GTX 1080 Ti, which is based on the latest NVidia Pascal architecture. GTX 1080 Ti affords 28 Streaming Multiprocessors (SM), each with 128 CUDA cores per SM, resulting in 3585 cores overall which enables 11.3TFLOPS for the single-precision calculation. The host system is Ubuntu 16.04 LTS with Intel Core i7-7700 @ 3.60Ghz, 64GB DDR4 memory, and a 512GB NVMe SSD. We use the latest version of the PGI Community Edition compiler 18.10[14], along with the CUDA 8 installation on our sys-
Table 1  Statistical results of the OpenACC implementation compared with the CUDA and the cluster version

|          | X₁ (fast) |          | X₂ (fast) |          | X₁ (slow) |
|----------|-----------|----------|-----------|----------|-----------|
|          | Mean      | Std      | Mean      | Std      | Mean      | Std      |
| OpenACC (GPU) | 1180.28   | 20.15    | 590.34    | 19.25    | 29.41     | 5.43     |
| CUDA (GPU)   | 1180.03   | 20.29    | 590.32    | 20.11    | 29.65     | 5.42     |
| StochKit (cluster) | 1180.56   | 20.10    | 589.68    | 19.92    | 29.76     | 5.43     |

Figure 3  Performance of OpenACC-based GSSA implementation

To comparatively analyze our OpenACC implementation, we use the performance evaluation results reported for both the CUDA and the CPU implementation in our previous work [7], where we used StochKit [15] on a Linux cluster for the CPU implementation. Table 1 shows statistical properties of our OpenACC implementation, together with the CUDA and the cluster implementations for simulating $2^{13}$ samples. The simulation results show that our OpenACC implementation works essentially the same as the CUDA and the cluster version.

Figure 3 shows the execution time of our OpenACC version of the GSSA implementation for the fast reversible isomerization process, together with the execution time of the CUDA version and the cluster version with varied number of simulation samples. The GPU version of the implementation, whether it is the OpenACC version or the CUDA version, shows about 3× better performance than the cluster version, which suggests that stochastic simulations are very well suited for parallelization on the GPUs due to its embarrassingly parallel nature of the computation.

Interestingly, the OpenACC implementation of the GSSA performs slightly better than the not fully optimized CUDA version. Although there is a lot of room for optimization improvements for the CUDA version of the GSSA, it is still very promising for the OpenACC version in that automated compiler optimizations can perform better than manually written CUDA implementations for certain applications in scientific computing.

5. Conclusions and Future Work

In this paper, we presented an OpenACC implementation of the stochastic simulation algorithm for biochemical reaction networks on the GPU accelerator. Specifically, we implemented the GSSA targeting one of the latest NVidia GPU architecture using OpenACC in a straightforward manner considering the level of parallelism. Our implementation shows almost equal or slightly better than the NVidia CUDA implementation, which can be a promising result for automated compiler optimizations compared against manual (and burdensome) efforts in terms of application performance. We plan to investigate a more general programming framework for stochastic simulations in order to cover a wide range of biochemical reaction network problems.

Acknowledgements

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] “OpenACC,” http://www.openacc-standard.org.
[4] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable Parallel Programming with CUDA,” Queue, vol.6, no.2, pp.40–53, March 2008.
[5] T. Hoshino, N. Martyama, S. Matsuoka, and R. Takaki, “CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application,” 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, pp.136–143, May 2013.
[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,” IETE Trans. Information and Systems, vol.101-D, no.3, pp.786–790, 2018.
[8] L. Dagum and R. Menon, “OpenMP: An Industry Standard API for Shared-Memory Programming,” IEEE Computational Science and Engineering, vol.5, no.1, pp.46–55, 1998.
[9] OpenACC-Standard.org, “The OpenACC Application Programming Interface v2.7,” 2018. https://www.openacc.org/sites/default/files/inlines-files/OpenACC2.7.pdf.
[10] “BLAS (Basic Linear Algebra Subprograms),” http://www.netlib.org/blas.
[11] C. Woolley, “Profiling and Tuning OpenACC Code,” GPU Technology Conference, 2012.
[12] 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.
[13] NVidia, “cuRAND,” https://docs.nvidia.com/cuda/curand/index.html.
[14] “PGI Community Edition,” https://www.pgroup.com/products/community.htm.
[15] H. Li, Y. Cao, L.R. Petzold, and D.T. Gillespie, “Algorithms and Software for Stochastic Simulation of Biochemical Reaction Systems,” Biotechnology Progress, vol.24, pp.56–61, 2008.