SNA + MC 2013, 04206 (2014)
DOI: 10.1051/snamc/201404206
© Owned by the authors, published by EDP Sciences, 2014

A comparative study of history-based versus vectorized Monte Carlo methods in the GPU/CUDA environment for a simple neutron eigenvalue problem

Tianyu Liu[1], Xining Du[1], Wei Ji[1], X. George Xu[1*], Forrest B. Brown[2]
[1]*Rensselaer Polytechnic Institute, Troy, New York, USA*
[2] *Los Alamos National Laboratory, Los Alamos, NM, USA*
[*]Corresponding Author, E-mail: xug2@rpi.edu

For nuclear reactor analysis such as the neutron eigenvalue calculations, the time consuming Monte Carlo (MC) simulations can be accelerated by using graphics processing units (GPUs). However, traditional MC methods are often history-based, and their performance on GPUs is affected significantly by the thread divergence problem. In this paper we describe the development of a newly designed event-based vectorized MC algorithm for solving the neutron eigenvalue problem. The code was implemented using NVIDIA's Compute Unified Device Architecture (CUDA), and tested on a NVIDIA Tesla M2090 GPU card. We found that although the vectorized MC algorithm greatly reduces the occurrence of thread divergence thus enhancing the warp execution efficiency, the overall simulation speed is roughly ten times slower than the history-based MC code on GPUs. Profiling results suggest that the slow speed is probably due to the memory access latency caused by the large amount of global memory transactions. Possible solutions to improve the code efficiency are discussed.

KEYWORDS: Monte Carlo, vectorized, event based, parallel computing, GPU, CUDA

I. Introduction

Radiation transport problems are frequently encountered in many applications including nuclear reactor analysis, medical imaging and radiation therapy. It is widely believed that Monte Carlo (MC) methods provide the most accurate results for radiation transport and are capable of dealing with complex geometry and physics models. However, MC simulations are often very time-consuming due to the large number of simulated histories required to reach satisfactory statistical precision. Despite the fast development of modern computers, it is still quite challenging to apply MC methods in routine nuclear reactor and medical physics calculations [1].

In recent years, graphics processing unit (GPU) and the relevant programming framework, especially NVIDIA's Compute Unified Device Architecture (CUDA) toolkit, have emerged as energy-efficient computing solutions and drawn wide attention in the supercomputing community. GPUs provide both tremendous computing power and the ease of use in parallel computing. In particular, GPU is suitable for applications with a large portion of work that can be carried out in parallel by multiple working threads. MC simulations are often embarrassingly parallel, meaning that individual histories can be computed simultaneously without communications between each other. It is thus natural to expect that by running MC simulations on GPUs, one can take advantage of the parallel computing power and reduce the simulation time accordingly.

Several groups have made preliminary efforts in developing GPU-based MC codes for neutron transport simulations. Heimlich et al. [2] studied the penetration probability of an incident neutron beam on a 1-D slab and observed a speedup factor of 15. Nelson and Ivanov [3] used more complex geometries and physics models to simulate the neutron transport, and reported a speedup of 11. Ding *et al* [4, 5] studied neutron eigenvalue problems using spherical and binary slab geometries, and observed speedup factors of 7.0 and 33.3, respectively, on NVIDIA Fermi GPUs compared to the same transport simulation running on CPUs.

In our previous work [4, 5], we developed the GPU-based MC code for a simple neutron eigenvalue problem using the history-based algorithms, where each thread on the GPU is used to deal with the entire history of one or more particles until they are absorbed or move out of the region of interest (ROI). Our results showed that this strategy is practical and the GPU-based code runs an order of magnitude faster than the corresponding CPU code.

However, history-based MC methods suffer from the so-called thread divergence issue on the GPU which can deteriorate the computing performance: On NVIDIA Fermi GPUs, every 32 threads are grouped into one warp, and the same instructions are executed for all 32 threads within each warp simultaneously. The result of this mechanism is that if threads within a warp are following different control flow, i.e. there are "if…else…" statements involved, then the divergent code segment will be executed sequentially, and this significantly reduces the parallel efficiency of the code. Unfortunately this is exactly the case for MC simulations where conditional statements are frequently used for event sampling. One solution to this issue is to use the so-called vectorized MC algorithms which can completely or partially eliminate the thread divergence, therefore facilitating more efficient GPU execution.

Vectorized MC algorithms were first developed in the 1980s, when Brown and Martin [6, 7, 8] implemented the algorithm on vector computers, e.g. Cyber-205 and Cray-1, and achieved great success. Today the commodity central processing units (CPUs), albeit often equipped with some vector components such as Streaming SIMD Extensions (SSE), adopt the superscalar architecture and do not work directly with the vectorized algorithm. On the contrary, NVIDIA GPUs are designed based on the single instruction multiple thread (SIMT) architecture [9], which has many similarities to the single instruction multiple data (SIMD) architecture of vector computers. While the hardware implementation is very different, the programming logic is nearly identical. It is thus speculated that previously developed vectorized algorithms could benefit from this hardware architecture and have the potential to run efficiently on GPUs. However, SIMT is a hybrid between vector processing and hardware threading while SIMD is particularly designed for vector processing. There are still substantial differences between these two architectures. As such, significant amount of work is needed to adapt the vectorized code to the GPU platform, making the study of vectorized MC methods on GPUs a non-trivial work task. As an early attempt, Bergmann et al. [10] investigated the vectorized MC algorithm on GPUs by solving a fixed-source problem in 2D geometry. They found that although the vectorized algorithm improves thread coherency, it does not outperform the conventional history-based algorithm on GPUs. In this work, we extended previous investigation to study vectorized algorithms for a different neutron transport problem, namely an eigenvalue problem in slab geometry.

This paper describes the use of ARCHER to analyze the performance of vectorized MC methods under the GPU/CUDA environment. We describe the code implementation and comparison of the performance of vectorized MC algorithms to that of the history-based algorithms in the GPU environment. This work is part of the project ARCHER (Accelerated Radiation-transport Computations in Heterogeneous EnviRonments) [11], which is designed as a comprehensive Monte Carlo software testbed using novel hardware and advanced programming models to speed up Monte Carlo calculations. Currently we employ ARCHER as a versatile research tool to evaluate the performance of Nvidia's GPU and Intel's coprocessor. In the long term, we envision ARCHER to be a suite of MC codes with the capability to gracefully scale on the exa-scale supercomputers. Figures 1 shows the design and long term vision of ARCHER.

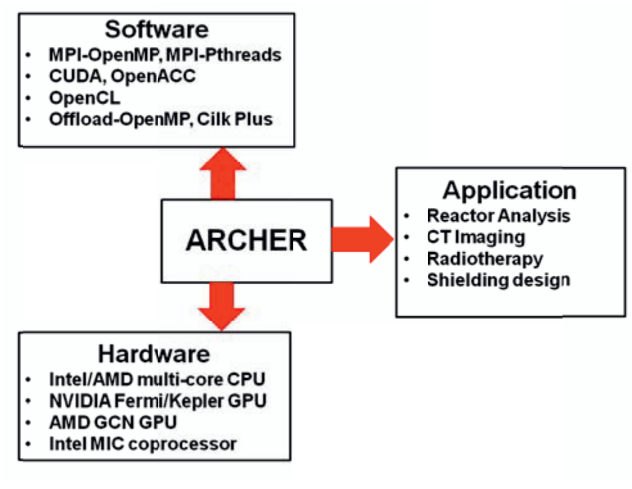

Figure 1: The vision of ARCHER. ARCHER is designed as a research tool to study MC simulations on a variety of hardware platforms including CPUs, GPUs and coprocessors. Different software tools including MPI, OpenMP, CUDA, etc. have been used in the development. ARCHER can be used in a variety of areas ranging from reactor analysis to radiotherapy and health physics.

II. Method

2.1 Neutron transport

The accurate prediction of multiplication factor is very important in nuclear reactor design and analysis. This is generally achieved by solving a k-eigenvalue problem for the reactor system under analysis. The multiplication factor is defined as the ratio of neutron populations between two successive fission generations. It directly reflects the system criticality as a function of geometry and material.

During MC simulation, the absorption process is simulated using non-analog method. Russian roulette and splitting technique is employed to ensure that the neutron always has an appropriate weight value. Collision and path-length estimators are used to evaluate eigenvalues in each generation.

2.2. Vectorized Monte Carlo Method

In MC simulations for neutron eigenvalue problem, each neutron will go through two different processes: the flight analysis and the collision analysis. In the flight analysis, we sample the neutron transport distance and move the particle to a new position. Then we check whether the neutron has reached the medium interface. If not, we perform the collision analysis and update the weight of the particle. Otherwise we continue to do another flight analysis until the sampled distance is less than the minimum distance to medium interface.

We implemented the vectorized MC algorithm following the method developed by Brown [6] for Cray vector computers. A simplified flow chart of the vectorized algorithm is shown in Figure 2. The basic idea it to keep two particle stacks for storing the neutrons being simulated in the current batch. One stack, called F, is used to store neutrons that will undergo the flight analysis. The other one, called C, is used to store neutrons that will undergo the collision analysis. In the beginning, we put all the initial neutrons into the F stack, and perform the flight analysis for all the neutrons. After distance sampling, those neutrons that will travel without crossing medium interfaces will be stored in the C stack for later collision analysis, and those that travel across medium interfaces will move to the interface position and stay in F stack for another flight analysis. This process is repeated until the F stack is empty, when the collision analysis is executed for all neutrons in the C stack. At this point, a shuffling operation is applied to stack C to remove neutrons that are out of ROI and only keep survived ones for the collision analysis. Neutrons with weight values below some critical value after collision will be removed following a sampling process. The survived neutrons then enter the next loop of analysis.

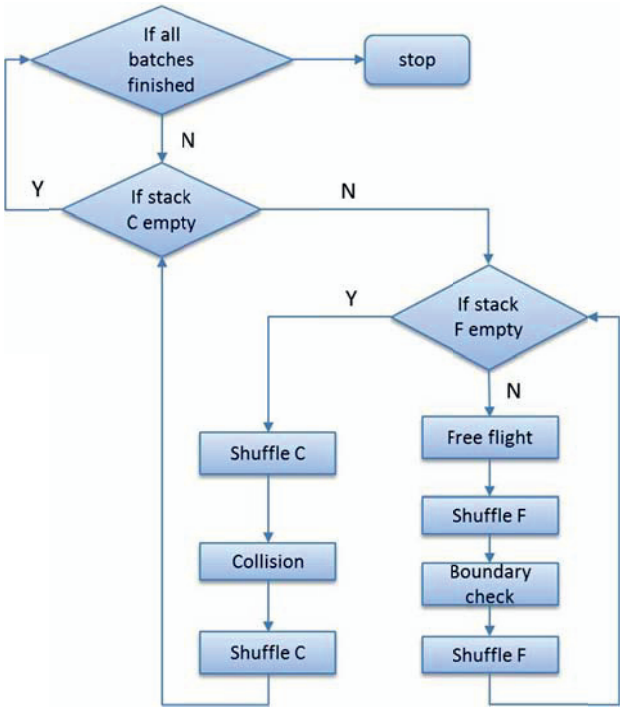

Figure 2: Simplified flowchart of GPU-based vectorized MC algorithm implemented in ARCHER.

By doing these iterative operations, we guarantee that all the neutrons being processed at the same time are undergoing the same physical events and involving the same sequence of instructions. This means that once the method is implemented on GPUs with each thread simulating one or more particles, all of the 32 threads within a warp will be executing the same instructions and the thread divergence does not occur. Different events are then executed iteratively in a sequential order until the storage stack becomes empty.

Note that with the event-based method, there is no one-to-one correspondence between the particle history and GPU thread, since different portions of one particle history may be executed by different threads for the events within that history.

2.3 GPU Implementation

We first developed a pure CPU code written in C, then ported the parallelizable modules (initialization of random numbers and fission sites, tracking of all neutron histories in a certain batch, etc.) of the code into ARCHER$_{GPU}$ using CUDA C[9]. Two different MC algorithms, both the history-based and event-based, were implemented on GPU, thus we have two versions of ARCHER$_{GPU}$ code.

We used Xorshift [12] pseudo random number generator (PRNG) for generating fast and high quality random numbers on the GPU. This PRNG algorithm is included in the CURAND library [13] provided by the CUDA kit, so it requires minimal development effort. Thrust library [14] was used to perform the shuffling operation, which is a key step in the vectorized MC algorithm.

Local variables including collision and path-length estimators were put on registers for fast access. Shared memory was used to store the partial sums of collision/path-length estimators for all of the threads within a block, which were then used to calculate the full sum by using the reduction technique. The neutron cross section data and geometrical parameters are shared by all of the threads and their values are not changed throughout the entire simulation, so these data were put in constant memory. Two neutron stacks storing the status data of neutrons were stored in global memory.

One change we made from the previous work is the usage of page-locked memories on the CPU side. We found that overuse of the page-lock memory could lead to a bandwidth bottleneck for a multi-GPU system, so in the vectorized MC code, we store all the intermediate data in the global memory on the device. The same change was made to our previously developed MC code, and test results were re-made by using the code after modification. This explains why the speedup factors of history-based MC algorithm shown in this paper are different from those in our previous paper [5].

The kernel block size was set to be 256, and the number of neutrons simulated by each thread was 100. The grid size was then determined by dividing the total number of neutrons to be handled for the current kernel by 25,600. The values of these parameters were chosen so that the GPU code performance was optimum for our particular setup.

III. Applications and Results

In this study, we considered a heterogeneous 1-D system that consists of alternately distributed fuel and moderator slabs. A total of 10 fuel slabs and 11 moderator slabs are modeled. For simplicity we use the one speed approximation in our MC implementation. Three physical processes, elastic scattering, fission and capture, are being considered for each neutron history in the simulations, where the last two are regarded as absorption. The cross sections of each reaction are set such that the resulting eigenvalue is close to one. Specifically, we use ΣF=0.034 cm-1, ΣA=0.08 cm-1, ΣT=0.1 cm-1, ν=2.5, Δx=3.8 cm for the fuel, and ΣA=0.0001 cm-1, ΣT=0.1 cm-1, Δx=30.0 cm for the moderator. The parameters were assigned such that the eigenvalues would finally be close to 1.

A total of 10^6 initial neutron histories and 1000 generations (the first 200 are inactive) were simulated by the CPU and GPU codes, respectively, to achieve convergence of eigenvalue and fission source distribution. Double precision floating point arithmetic was used for guaranteeing the computation precision.

For the performance benchmark, we used a work-station equipped with an Intel Xeon X5650 2.66GHz CPU and a NVIDIA Tesla M2090 GPU card with 6GB global memory. Three versions of the code were tested: ARCHER$_{CPU}$ was run in serial mode using a single thread on the CPU, while history-based ARCHER$_{GPU}$ and vectorized ARCHER$_{GPU}$ were run on the GPU card.

In Table 1, we show the running time of three different codes and the speed up factors relative to the CPU code.

Code	**Computation time [sec]**	**Speedup**
ARCHER$_{CPU}$	6077.5	1
ARCHER$_{GPU}$ (history-based)	208.1	29.2
ARCHER$_{GPU}$ (vectorized)	2278.9	2.7

Table 1. Performance comparison between different transport codes in ARCHER.

The GPU-based vectorized Monte Carlo code, ARCHER$_{GPU}$ (vectorized), was found to be slower than its conventional GPU counterpart, ARCHER$_{GPU}$ (history-based), by a factor over 10. To find out the cause of the downgraded performance, the GPU execution statistics per neutron generation were collected and analyzed by using a profiling tool NVPROF [15]. In Figure 8 we show the control flow efficiency of each kernel function for the history-based and vectorized algorithm. Control flow efficiency, defined as

Control flow efficiency = {Thread Instructions Executed} / {Instructions Executed} / {Warps Size},

is a measure of how many threads are active for each instruction. In the ideal case when all the threads within a warp execute the same instruction, this number will be 100%. The more the thread divergence occurs, the lower this number is. From Figure 3, it can be seen that overall the control flow efficiency is 2-3 times higher for vectorized MC kernel functions compared with the history-based MC kernels. This means that the occurrence of thread divergence is significantly reduced in vectorized GPU-based MC codes.

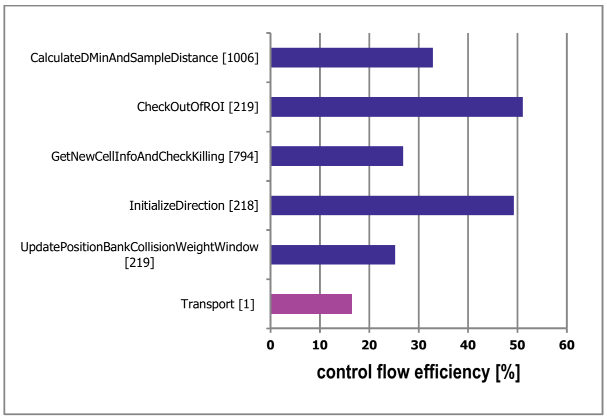

Figure 3: Control flow efficiency of kernel functions. Magenta and navy bars represent the data for history-based and vectorized ARCHER$_{GPU}$ codes, respectively. Numbers in the square brackets denote kernel launch times.

However, the advantage of higher control flow efficiency is completely offset by the highly increased global memory transaction, as is illustrated in Figure 4. Unlike in the conventional code where neutron attribution data such as position, direction, energy, weight, etc. are created and consumed in the fast on-chip registers, in the vectorized code, they have to be frequently read from and written to the slow off-chip global memory, which is known to have high access latency. For the neutron eigenvalue problem considered in this study, the total global memory throughput per neutron generation of the vectorized code is on the order of 200 GB, which is ~60 times larger than that of the conventional code. The dramatically increased number of global memory transactions causes large amount of latencies on the GPU and makes the vectorized MC code much slower than the conventional one, although the former gives much better control flow efficiency.

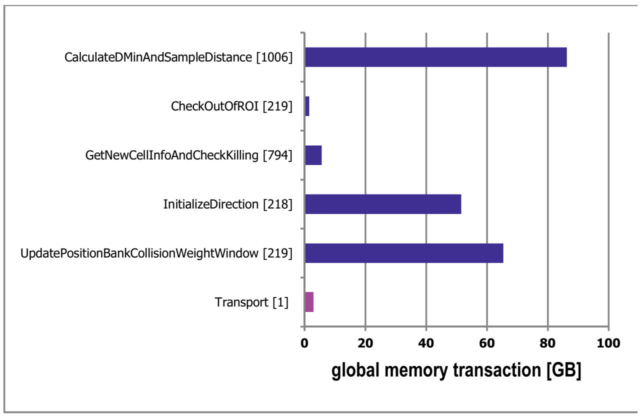

Figure 4: Global memory throughput of kernel functions. Magnetic and navy bars represent the data for history-based and vectorized ARCHER$_{GPU}$ codes, respectively. Number in the square bracket denotes the kernel launch times.

Based on the test runs and profiling results, we can draw the preliminary conclusion that vectorized MC algorithm is probably not well suited for running on modern GPUs. The main reason is the high latency of global memory access associated with the vectorized algorithm deteriorates the GPU performance. Frequent memory reading/writing is to a large extent intrinsic to the vectorized algorithm, so latency is most likely to continue to be a major issue for any effort of porting vectorized MC code to GPUs.

There are techniques that can be used to possibly alleviate the global memory latency problem and they will be investigated in our future studies:

1. In the flight analysis step, we keep executing the flight kernel until all of the neutrons enter the collision stack. As this process goes, the number of active neutrons simulated in the flight kernel is continuing to decrease and it could be well below one million for the last several flight kernel executions. In our GPU code, we always use a block size of 256 and let each thread simulate 100 neutrons, so if the total number of neutrons is only, say, tens of thousands, there are merely several hundred active threads for the GPU and they are much less than the maximal active concurrent threads on the Tesla M2090 card, which is 24576. The stream processors (SM) in GPU are not fully occupied, resulting in a very low efficient GPU usage. One possible solution is to adjust the block size and the number of neutrons per thread dynamically according to the current total number of neutrons being simulated to guarantee enough number of concurrent threads. Another way to improve the GPU occupancy is to use stack-driven vectorized MC, where one always chooses the stack with the largest number of particles to process. This technique can avoid the kernel launch for events with relatively small size stacks and therefore reduce the possibilities of inefficient GPU kernel executions.

2. Because the vectorized algorithms are effective in reducing thread divergence, it may be beneficial to address the global memory access problem directly. There are many "tried and true" programming techniques to hide the latencies for memory access, and some of those techniques may be useful for GPUs. For example, having several stacks for both free-flight and collision operations may permit fetching one stack from memory while the GPUs are operating on another stack. Techniques such as prefetching, read-ahead / write-behind, asynchronous access, etc., have proven effective for conventional CPUs and may be effective in latency hiding for the GPU global memory access. These programming techniques, however, further complicate the Monte Carlo algorithms and may be highly dependent on relative speeds of specific hardware.

IV. CONCLUSIONS

In this work, we have used the ARCHER testbed to investigate the performance of vectorized MC methods in the GPU/CUDA environment. We applied the vectorized MC method to a neutron eigenvalue problem and compared its performance with the conventional MC method on GPUs as well as those on CPUs. We found that the vectorized MC code indeed reduced the occurrence of thread divergence, which is known as a challenge in running MC simulations efficiently on GPU's SIMT architecture, and greatly increased the control flow efficiency. However, we also found that the new algorithm was actually ten times slower than the conventional MC algorithm on GPUs, mainly due to the increased number of global memory transactions. Our preliminary conclusion is that the vectorized MC algorithm, as implemented in this study, is not well suited for GPUs. We are currently testing other hardware designs including the Intel Xeon Phi coprocessor. We then proposed several directions in which further work can be done to enhance the performance of vectorized algorithm: (1) more flexible arrangement of block size and thread working load, (2) the hybrid algorithm combining both history-based and event-based MC methods, and (3) programming techniques to hide memory latency by overlapping computations with memory access. These future works, once done, should provide more insight into the role of vectorized MC algorithm in future peta-scale and exa-scale high performance computers.

Acknowledgment

This work was funded in part by the grant from National Institute of Biomedical Imaging and Bioengineering (R01EB015478).

References

1. Pratx G and Xing L, "GPU computing in medical physics: a review", *Med. Phys.* 38 2685–97 (2012).
2. A. Heimlich, A. C. A. Mol, C. M. N. A. Pereira, "GPU-Based High Performance Monte Carlo Simulation in Neutron Transport and finite differences heat equation evaluation", 2009 International Nuclear Atlantic Conference, Rio de Janeiro, RJ, Brazil, September 27-October 2, 2009 (2009).
3. A. G. Nelson, K. N. Ivanov, "Monte Carlo methods for neutron transport on graphics processing units using CUDA", PHYSOR 2010 – Advances in Reactor Physics to Power the Nuclear Renaissance, Pittsburgh, Pennsylvania, USA, May 9-14, 2010 (2010).
4. A. Ding, T. Liu, C. Liang, W. Ji, M. S. Shephard, X. G. Xu, F. B. Brown. "Evaluation of speedup of Monte Carlo calculations of simple reactor physics problems coded for the GPU/CUDA environment", ANS Mathematics & Computation Topical Meeting, Rio de Janeiro, RJ, Brazil, May 8-12, 2011(2011).
5. T. Liu, A. Ding, W. Ji, X. G. Xu, C. D. Carothers, F. B. Brown, "A Monte Carlo neutron transport code for eigenvalue calculations on a dual-GPU system and CUDA environment", *Physor 2012 Advances in Reactor Physics*, Knoxville, TN, USA, April 15-20, 2012.
6. F. B. Brown, W. R. Martin, "Monte Carlo methods for radiation transport analysis on vector computers", *Progress in Nuclear Energy*, **Vol. 14, No. 3**, pp. 269-299 (1984).
7. W. R. Martin, F. B. Brown, "Status of vectorized Monte Carlo for particle transport analysis", *The International Journal of Supercomputer Applications*, **Vol. 1, No. 2**, pp. 11-32 (1987).
8. W. R. Martin, "Successful vectorization-reactor physics Monte Carlo code", *Computer Physics Communications*, **Vol. 57, No. 1-3**, pp. 68-77 (1989).
9. "CUDA C programming guide", http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, NVIDIA (2013).
10. R. M. Bergmann, J. L. Vujic, N. A. Fischer, "2D Mono-Energetic Monte Carlo Particle Transport on a GPU", ANS Winter Meeting Transactions, November 2012.
11. X.G. Xu, T. Liu, L. Su, X. Du, M.J. Riblett, W. Ji. "An Update of ARCHER, a Monte Carlo Radiation Transport Software Testbed for Emerging Hardware Such as GPUs". 2013 American Nuclear Society Annual Meeting, Atlanta, GA, June 16-20, 2013.
12. M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3-30 (1998).
13. "CUDA Toolkit 5.5 CURAND Guide", NVIDIA (2013).
14. "CUDA Toolkit 5.5 Thrust Guide", http://docs.nvidia.com/cuda/thrust/index.html, NVIDIA (2013).
15. "Profiler User's Guide", NVIDIA (2013).