EVALUATION OF VECTORIZED MONTE CARLO ALGORITHMS ON GPUS FOR A NEUTRON EIGENVALUE PROBLEM

Xining Du, Tianyu Liu, Wei Ji, X. George Xu*
Nuclear Engineering Program,
Rensselaer Polytechnic Institute
Troy, NY 12180, USA
dux3@rpi.edu; liut4@rpi.edu; jiw2@rpi.edu; xug2@rpi.edu

Forrest B. Brown
Monte Carlo Codes Group
Los Alamos National Laboratory
Los Alamos, NM 87545
fbrown@lanl.gov

ABSTRACT

Conventional Monte Carlo (MC) methods for radiation transport computations are "history-based", which means that one particle history at a time is tracked. Simulations based on such methods suffer from thread divergence on the graphics processing unit (GPU), which severely affects the performance of GPUs. To circumvent this limitation, event-based vectorized MC algorithms can be utilized. A versatile software testbed, called ARCHER - Accelerated Radiation-transport Computations in Heterogeneous EnviRonments - was used for this study. ARCHER facilitates the development and testing of a MC code based on the vectorized MC algorithm implemented on GPUs by using NVIDIA's Compute Unified Device Architecture (CUDA). The ARCHER$_{GPU}$ code was designed to solve a neutron eigenvalue problem and was tested on a NVIDIA Tesla M2090 Fermi card. We found that although the vectorized MC method significantly reduces the occurrence of divergent branching and enhances the warp execution efficiency, the overall simulation speed is ten times slower than the conventional history-based MC method on GPUs. By analyzing detailed GPU profiling information from ARCHER, we discovered that the main reason was the large amount of global memory transactions, causing severe memory access latency. Several possible solutions to alleviate the memory latency issue are discussed.

Key Words: vectorized Monte Carlo method, neutron transport, GPU, CUDA

1. INTRODUCTION

Radiation transport computation has been widely applied to many scientific and engineering areas to predict the radiation particle behavior. It has played important roles in nuclear reactor design, medical imaging and radiation therapy. To perform radiation transport simulations, one can either use deterministic algorithms based on some simplified physics models, or rely on the stochastic Monte Carlo (MC) method [1]. While the MC method provides the most accurate predictions and is capable of handling complex geometry and physics models, it is time-consuming due to the large number of histories required to reduce the statistic error. Despite impressive advancement in CPU clock speed, MC methods are still not fast enough for routine applications in nuclear reactor engineering and medical physics.

Recently developed graphics processing unit (GPU) and the relevant programming framework, especially NVIDIA's CUDA toolkit, provide tremendous computing power and the ease of use in parallel computing [2]. The GPU hardware is particularly suitable for applications having a large portion of work that can be carried out in parallel. MC simulations are inherently parallel [2], and individual particle histories can be computed concurrently. Therefore it is natural to expect that by running MC simulations on GPUs, one can take advantage of the computing power of these parallel devices and significantly reduce the time to perform the simulation.

Several groups have made preliminary efforts in developing GPU-based MC codes for neutron transport simulations. Heimlich et al. [3] studied the penetration probability of an incident neutron beam on a 1-D slab and observed a speedup factor of 15. Nelson and Ivanov [4] used more complex geometries and physics models to simulate the neutron transport, and reported a speedup of 11. In our previous studies [5, 6], we performed a similar study of the eigenvalue problem 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.

We adopted the conventional history-based MC algorithm in our previous work [5]. Each processing unit is used to deal with the entire history of one or more particles until all of the particles are absorbed or move out of the region of interest (ROI). In the CUDA framework, the algorithm can be implemented with one thread being a fundamental processing unit. Our results showed that this strategy is practical and the GPU code is an order of magnitude faster than the counterpart CPU code.

However, the inherent thread divergence issue with the history-based MC code on GPUs has a significant impact on the GPU performance and hinders the opportunity to exploit the GPU computing power. In the NVIDIA Fermi architecture, every 32 threads are grouped into one warp, and the same instructions are executed for all 32 threads in the warp simultaneously. The result of this mechanism is that if threads within a warp are following different control flow, i.e. there are some "if…else…" statements involved, which is exactly the case for MC simulations, then the divergent code segment will be executed sequentially, and this significantly reduces the parallel efficiency of the code. One solution is to use so-called vectorized MC algorithms in which the thread divergence is completely or partially eliminated, therefore facilitating more efficient GPU execution.

Vectorized MC algorithms were first developed in the 1980s, when Brown and Martin [7, 8, 9] 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, 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.

For investigating issues related to emerging high-performance computing, we have recently launched an ambitious project to develop a software testbed called **ARCHER** (**A**ccelerated **R**adiation-transport **C**omputations in **H**eterogeneous **E**nvi**R**onments). 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.

2. METHODOLOGY DESCRIPTIONS

2.1 Eigenvalue Problem Descriptions

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. Using MC, we simulate the physical transport processes of a large number of neutrons from generation to generation and calculate the eigenvalue based on the numbers of neutrons.

The geometry considered here is 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 Σ_F=0.034 cm^{-1}, Σ_A=0.08 cm^{-1}, Σ_T=0.1 cm^{-1}, ν=2.5, Δx=3.8 cm for the fuel, and Σ_A=0.0001 cm^{-1}, Σ_T=0.1 cm^{-1}, Δx=30.0 cm for the moderator.

In the 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 the MC method for the 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.

The flow chart of vectorized MC method as implemented in ARCHER is shown in Figure 1. We 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.

```
Loop for Batches 1 … N
.
.       While  length(C) > 0
.       .
.       .       While  length(F) > 0
.       .       .       Free Flight
.       .       .       Shuffle F
.       .       .       Boundary Crossing Check
.       .       .       Shuffle F
.       .       . . . . .
.       .
.       .       Shuffle C
.       .       Collision
.       .       Shuffle C
.       . . . . .
.
. . . .
```

Figure 1. 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 using the GPU-specific language CUDA C [11]. This was done for both the conventional and vectorized Monte Carlo methods. The pseudo-random number generator we adopted was Mersenne Twister [12] for the ARCHER$_{CPU}$ code and XORWOW [13] for the ARCHER$_{GPU}$ code.

2.3.1. GPU memory allocation

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 [6].

2.3.2 GPU thread arrangement

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.

3. RESULTS AND DISCUSSIONS

In Table I, we show the running time of three different codes and the speed up factors relative to the CPU result. The ARCHER$_{CPU}$ code was tested on an Intel Xeon X5650 2.66GHz CPU with 13G DDR3 memory. Only a single CPU thread was used and the code was run in sequential mode. The ARCHER$_{GPU}$ codes were tested on a NVIDIA Tesla M2090 GPU card.

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

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

The GPU-based vectorized Monte Carlo code, ARCHER$_{GPU}$ (vectorized), was found to be slower than its conventional GPU counterpart, ARCHER$_{GPU}$ (conventional), 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 [14]. Figure 2 displays the warp execution efficiency data of each kernel function, which is defined as the average number of active threads in a warp divided by the total number of threads in a warp (32). The magenta and green bars represent the GPU profiling data for conventional and vectorized algorithms, respectively. Numbers in the square brackets denote the number of times that each kernel is launched. The kernels that are used to initialize fission sites and neutron weights prior to the simulation of each generation in the conventional and vectorized codes are functionally the same; hence a similar efficiency. Except for that, all the kernels of the vectorized code have a higher efficiency than the single large transport kernel of the conventional code, because of the effectively decreased occurrence of divergent branches.

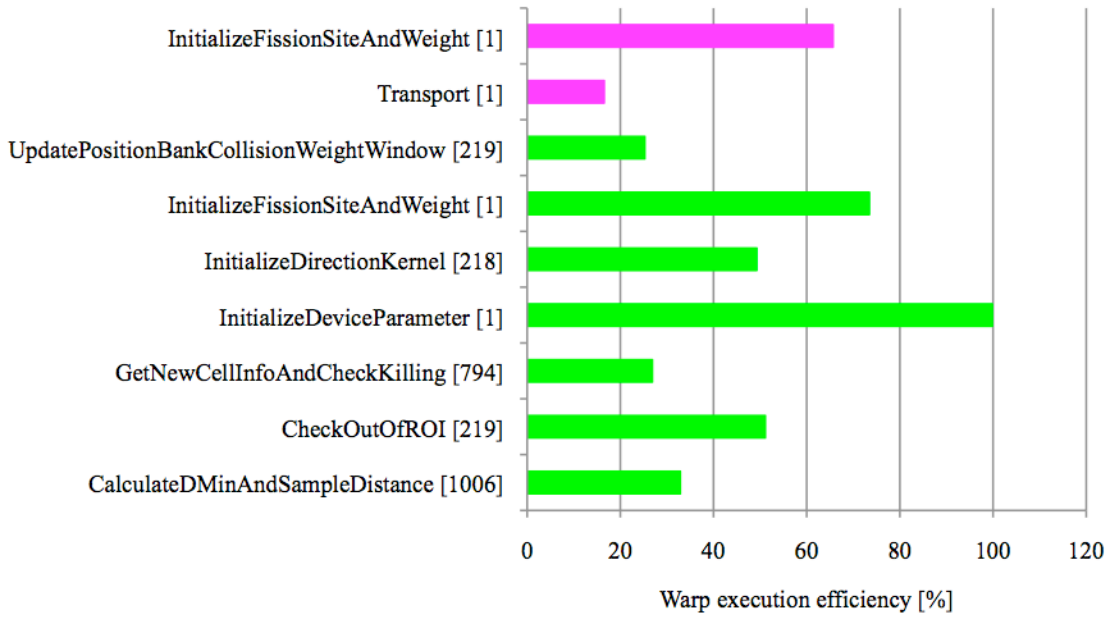

Figure 2. Warp execution efficiency of kernel functions. Magenta and green bars represent the data for history-based and vectorized ARCHER$_{GPU}$ codes, respectively. Numbers in the square brackets denote the number of times that the kernel is launched.

However, the advantage of higher warp execution efficiency is completely offset by the highly increased global memory transaction, as is illustrated in Figure 3. Following the same convention, we use magenta and green bars to represent data for conventional and vectorized algorithms, respectively, and indicate the kernel launch counts in the square brackets after each kernel. Unlike in the conventional code where the neutron attribute data such as position, direction, energy, weight, etc. are created and consumed in the fast on-chip register space, 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 a high access latency. For the 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 better warp execution efficiency.

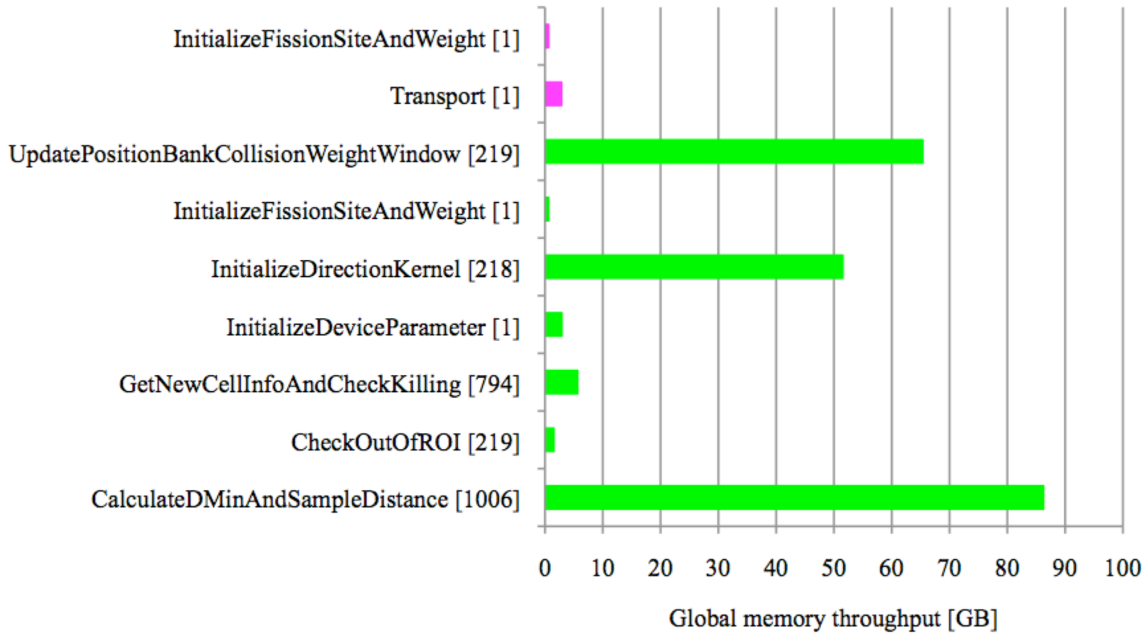

Figure 3. Global memory throughput of kernel functions. Magnetic and green bars represent the data for history-based and vectorized ARCHER$_{GPU}$ codes, respectively. Number in the square bracket denotes the times that the kernel is launched.

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 and the main reason is the high latency of global memory access. The requirement for 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 possibilities to alleviate the global memory latency problem and improve the performance of the GPU code. Here we propose several approaches:

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. We plan to implement this feature in the newer version of our code.

2. An alternative way is to consider a hybrid algorithm which mixes the history-based and event-based methods. Specifically, we can use the event-based vectorized MC as the main framework to simulate generations of neutrons. Within each generation, we follow the strategy of the vectorized MC algorithm to do the flight analysis until the number of neutrons to be simulated falls below a critical value. From that point on, we switch to the history-based algorithm to simulate the residue neutrons. By doing this, one can not only avoid the inefficient vectorized kernel executions for small number of neutrons, but also reduce the amount of global memory transactions for loading and writing neutron data. Other neutrons that already enter the collision stack will continue to be processed by the collision kernel in the event-based algorithm. Then we join the neutrons that survive from two separate paths and use them for the next generation to be simulated. The hope is that by combining the two algorithms, one can take advantage of the most efficient parts of two different algorithms and find a better strategy for using the GPU resources in a more optimized way. This might need to make substantial changes to our algorithm and it is subjected to future investigations.

3. 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.

4. 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 ARCHER$_{GPU}$ (vectorized) indeed reduced the occurrence of thread divergence - a challenge in running MC simulations efficiently on GPU's SIMT architecture. Thus the warp execution efficiency was greatly increased due to the usage of the event-based vectorized algorithm. 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.

ACKNOWLEDGMENTS

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

REFERENCES

1. B.L. Kirk, "Overview of Monte Carlo radiation transport codes", *Radiation Measurements*, Volume 45, Issue 10, pp. 1318-1322 (2010).
2. Pratx G and Xing L, "GPU computing in medical physics: a review", *Med. Phys.* 38 2685–97 (2012).
3. 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).
4. 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).
5. 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).
6. 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.
7. 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).
8. 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).
9. W. R. Martin, "Successful vectorization-reactor physics Monte Carlo code", *Computer Physics Communications*, **Vol. 57, No. 1-3**, pp. 68-77 (1989).

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. “CUDA C programming guide”, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, NVIDIA (2012).
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. NVIDIA, “CUDA Toolkit 5.0 CURAND Guide”, (2012).
14. NVIDIA, “Profiler User’s Guide”, (2012).