PHYSOR 2012 – Advances in Reactor Physics – Linking Research, Industry, and Education
Knoxville, Tennessee, USA, April 15-20, 2012, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2012)

A MONTE CARLO NEUTRON TRANSPORT CODE FOR EIGENVALUE CALCULATIONS ON A DUAL-GPU SYSTEM AND CUDA ENVIRONMENT

Tianyu Liu, Aiping Ding, Wei Ji, and X. George Xu
Nuclear Engineering and Engineering Physics,
Rensselaer Polytechnic Institute
Troy, NY 12180, USA
xug2@rpi.edu

Christopher D. Carothers
Department of Computer Science,
Rensselaer Polytechnic Institute (RPI)

Forrest B. Brown
Los Alamos National Laboratory (LANL)

ABSTRACT

Monte Carlo (MC) method is able to accurately calculate eigenvalues in reactor analysis. Its lengthy computation time can be reduced by general-purpose computing on Graphics Processing Units (GPU), one of the latest parallel computing techniques under development. The method of porting a regular transport code to GPU is usually very straightforward due to the “embarrassingly parallel” nature of MC code. However, the situation becomes different for eigenvalue calculation in that it will be performed on a generation-by-generation basis and the thread coordination should be explicitly taken care of. This paper presents our effort to develop such a GPU-based MC code in Compute Unified Device Architecture (CUDA) environment. The code is able to perform eigenvalue calculation under simple geometries on a multi-GPU system. The specifics of algorithm design, including thread organization and memory management were described in detail. The original CPU version of the code was tested on an Intel Xeon X5660 2.8GHz CPU, and the adapted GPU version was tested on NVIDIA Tesla M2090 GPUs. Double-precision floating point format was used throughout the calculation. The result showed that a speedup of 7.0 and 33.3 were obtained for a bare spherical core and a binary slab system respectively. The speedup factor was further increased by a factor of ~2 on a dual GPU system. The upper limit of device-level parallelism was analyzed, and a possible method to enhance the thread-level parallelism was proposed.

Key Words: Monte Carlo, eigenvalue calculation, GPU, CUDA, neutron transport, reactor analysis.

1. INTRODUCTION

Monte Carlo simulation provides the most accurate solution of the Boltzmann neutron transport equation in heterogeneous, 3D media such as a nuclear reactor core. However, the large computation time that Monte Carlo requires to obtain results of acceptable precision has severely limited its uses in the design and analysis of nuclear reactor systems. The so-called “Kord Smith

Challenge" highlights the interest by the reactor physics community in performing a full-core Monte Carlo analysis [1]. Smith predicted that a full-core Monte Carlo calculation would take 5,000 hours on a 2-GHz PC and, according to Moore's Law, could not be performed in less than one hour until the year 2030. Martin re-analyzed the topic at the 2007 ANS Mathematics & Computation Conference and concluded that it would be 2019 when such a full reactor core calculation could be accomplished in one hour by making use of a 1500-core processor at that time [2]. To monitor the Monte Carlo computational performances, a benchmark test model of a simplified Pressured Water Reactor (PWR) core assembly was subsequently proposed by Hoogenboom and Martin in 2009 [3] and later revised by Hoogenboom et al in 2010 [4].

The exascale massively parallel high-performance computers, expected to arrive by the end of this decade, are capable of a peak performance on the order of 10^{18} FLOPS and it may be possible to accelerate Monte Carlo based full-core reactor analysis to a level of minutes or even seconds. The bad news, however, is that the next-generation hardware architecture — and the associated software — is still very much undefined. The high-performance computing (HPC) community has taken the "co-design" approach to a number of emerging technologies in both inter-node and intra-node software models (MPI, openMP, pThreads, CUDA, openCL, etc) [5]. The hardware industry is moving toward heterogeneous systems, where GPUs and CPUs are integrated to perform general purpose computing tasks. As a parallel processor, the GPU excels in tackling large number of neutron transport histories in a Monte Carlo code that can be calculated simultaneously. The world's most powerful supercomputer in 2010 — the Tianhe-1 system — achieved a peak performance of 2.507 petaFLOPS by harvesting the power of 14,336 CPUs and 7,168 Tesla M2050 GPUs. In October 2011, ORNL announced the plan to upgrade its Jaguar computer from 2.3 to 20 petaFLOPS by using NIVIDIA's Tesla GPUs. The new system, which will be known as Titan, is hoped to regain the position as the world's more powerful supercomputer. Therefore, GPUs and the associated CUDA software environment by NIVIDIA will likely be visible in future exascale computers.

Rapid changes in HPC are bringing an unprecedented uncertainty about the compatibility of application software packages. There is a growing concern that existing production Monte Carlo codes may not be able to take advantage of the petascale and exascale computer systems [6]. Brown observes, during an invited lecture for the Monte Carlo 2010 Conference in Tokyo, that the existing production Monte Carlo codes such as MCNP were not optimized for large-scale parallel computing involving the GPU/CUDA platform and that it may be prudent to develop an entirely new code [6].

Since 2009 several groups have made preliminary effort to develop GPU-based MC codes for transport simulations. Heimlich et al. [7] calculated the penetration probability of a beam of neutrons incident upon a 1-D slab, and observed a speedup of 15. Nelson and Ivanov [8] adopted more complicated geometries and physical models to simulate the neutron transport, and reported a speedup of 11 using double-precision floating point calculations. Xu et al. [9] studied a fixed source problem using 1-D binary slab geometry and an eigenvalue problem using 1-D homogeneous slab geometry, and observed a speedup of 29.2 and 8.5.

On the other hand, for photon simulations, the radiation treatment planning community has been much intrigued by the possibility of performing sub-minute MC dose calculations in a busy clinic setting [10,11,12]. High speedups were reported when the GPU codes were compared with general-purpose MC codes running on CPUs. MC-GPU by Badal et al. [13] was aimed primarily at radiography simulation and was reported to be 110 times faster than PENELOPE [14], a package to simulate electron-photon transport. GPUMCD by Hissoiny et al. [15] was used for dosimetric calculation and was over 900 times faster than EGSnrc [16], another package for coupled electron-photon transport, and over 200 times faster than DPM, an optimized code for radiotherapy treatment planning dose calculations. gCTD by Jia et al. [17] was used to quantify radiation dose received by patients during a CT scan and was found to be 400 times faster compared to the EGSnrc code in a simplistic water phantom geometry.

Generally for the regular MC transport simulation of neutrons or photons, the way to translate a CPU code into a GPU counterpart is very straightforward. First, the CPU evenly distributes the total task to all the threads, which are the GPU execution units. Then, each thread independently performs calculation in exactly the same way a serial CPU code does except that the number of iterations is reduced due to the parallelism offered by the GPU. Finally, after the work on the GPU is done, the results are sent back to the CPU. Problems that are parallelizable this way are often addressed by parallel computing community as “embarrassingly parallel” problems, meaning it takes little or no effort to coordinate between threads.

This convenience is not fully enjoyed by the GPU-based MC eigenvalue calculation which is a fundamental task in reactor design and analysis. In such a problem, the neutrons are simulated on a generation-by-generation basis. The number of histories to be simulated in each generation is varying due to the random nature of MC, implying that the number of threads to execute the simulation task is also varying. Since thread reorganization can only be performed on the CPU, one cannot simply devolve everything to GPU and wait for the result to come out. Instead, it is the programmer’s responsibility to (1) reassign threads to the GPU each time the per-generation calculation is finished, (2) send intermediate data from the device GPU to the host CPU, process them properly and send them back to the device for calculating the next generation.

To investigate these issues, we developed a GPU-based MC eigenvalue calculation code that can be applied to a multi-GPU system to achieve both thread-level and device-level parallelism. A dual-GPU system was used in order to test the scalability of the code.

2. MATERIALS AND METHODS

2.1. Software Development

2.1.1. Algorithm of eigenvalue calculations performed on a CPU

The effective multiplication factor, also known as eigenvalue, is defined as the ratio of the number of fission neutrons in one generation to that in the previous generation. It directly reflects the system criticality as a function of geometry and material. The flowchart of MC eigenvalue calculation is shown in Figure 1. The algorithm is composed of two iterations: the outer iteration is done over different fission source generations, while the inner iteration is done over different histories of a certain generation and can be regarded as an equivalent to the regular history-based transport problem. It should be noted that the output of each generation constitutes the input data of the next generation. This was carefully taken care of in the GPU programming.

Our study concentrated on calculating eigenvalues under two typical geometrical conditions: a bare spherical core and a binary slab system. The bare spherical core was made of homogeneous fissile material, and the binary slab system of fuel and moderator that were distributed alternately— a simplified representation of the fuel pins submerged in water moderator in a light water reactor (LWR). One speed approximation was made in our neutron transport physical model which included elastic scattering, fission and capture, where the last two interactions were collectively regarded as absorption. For the bare spherical core model, the parameters were: $\Sigma f=0.01$ cm^{-1}, $\Sigma a=0.02$ cm^{-1}, $\Sigma t=0.1$ cm^{-1}, $\nu=2.5$, R= 74.0116 cm. For the binary slab model, the parameters of the fuel slabs were: $\Sigma f=0.01$ cm^{-1}, $\Sigma a=0.02$ cm^{-1}, $\Sigma t=0.1$ cm^{-1}, $\nu=2.5$, $\Delta x=3.8$ cm, and the parameters of the moderator slabs were: $\Sigma a=0$ cm^{-1}, $\Sigma t=0.1$ cm^{-1}, $\Delta x=10.0$ cm. The parameters were assigned such that the eigenvalues would finally be close to 1. The absorption process was simulated using non-analog method. Weight-window technique composed of splitting and Russian roulette was employed to ensure the neutron always had an appropriate weight value. Collision and path-length estimators (kcol, kpath) were used to evaluate eigenvalues in each generation. Relative standard deviation was below 1%. The convergence of eigenvalue and fission source distribution was guaranteed by simulating a total of 1000 generations, the first 200 out of which were made inactive. The CPU code was written in C++ and tested on a desktop with 9 GB Random Access Memory (RAM) and an Intel Xeon X5660 2.8GHz CPU in a single-thread and single-core mode.

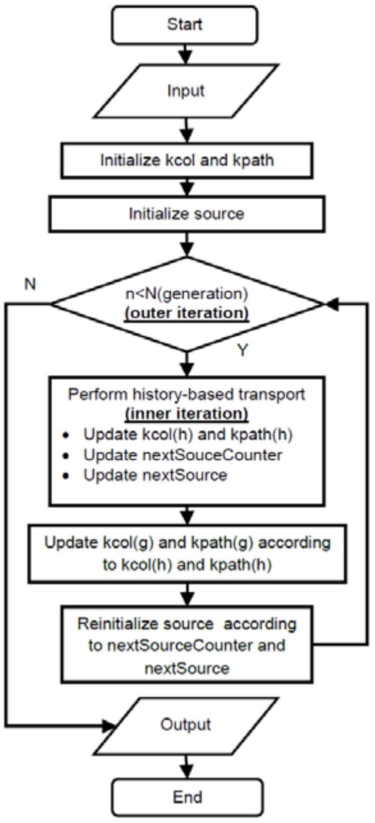
Start
Input
Initialize kcol and kpath
Initialize source
n<N(generation)
(outer iteration)
N
Y
Perform history-based transport
(inner iteration)
• Update kcol(h) and kpath(h)
• Update nextSouceCounter
• Update nextSource
Update kcol(g) and kpath(g) according to kcol(h) and kpath(h)
Reinitialize source according to nextSourceCounter and nextSource
Output
End

Figure 1 Flowchart of the CPU-based eigenvalue calculation program. N(generation) is the number of histories to be simulated in each generation and is a randomly varying number. The inner iteration is equivalent to the regular transport problem. For brevity, its

particulars are not entirely shown. kcol(h) and kpath(h) are intermediate variables used to update the per-generation eigenvalues kcol(g) and kpath(g). nextSourceCounter counts the number of fission neutrons generated that will be used as the source for the next generation.

2.1.2. Algorithm of eigenvalue calculations performed on a dual GPU system

The GPU code was adapted from the CPU version, taking into account the specialties and intricacies of MC eigenvalue calculations briefly mentioned in the introduction section. Figure 2 exhibits the flowchart of the GPU code which includes the important steps required for GPU programming. Two crucial factors were carefully considered, namely, thread organization and memory management, which are discussed below in detail.

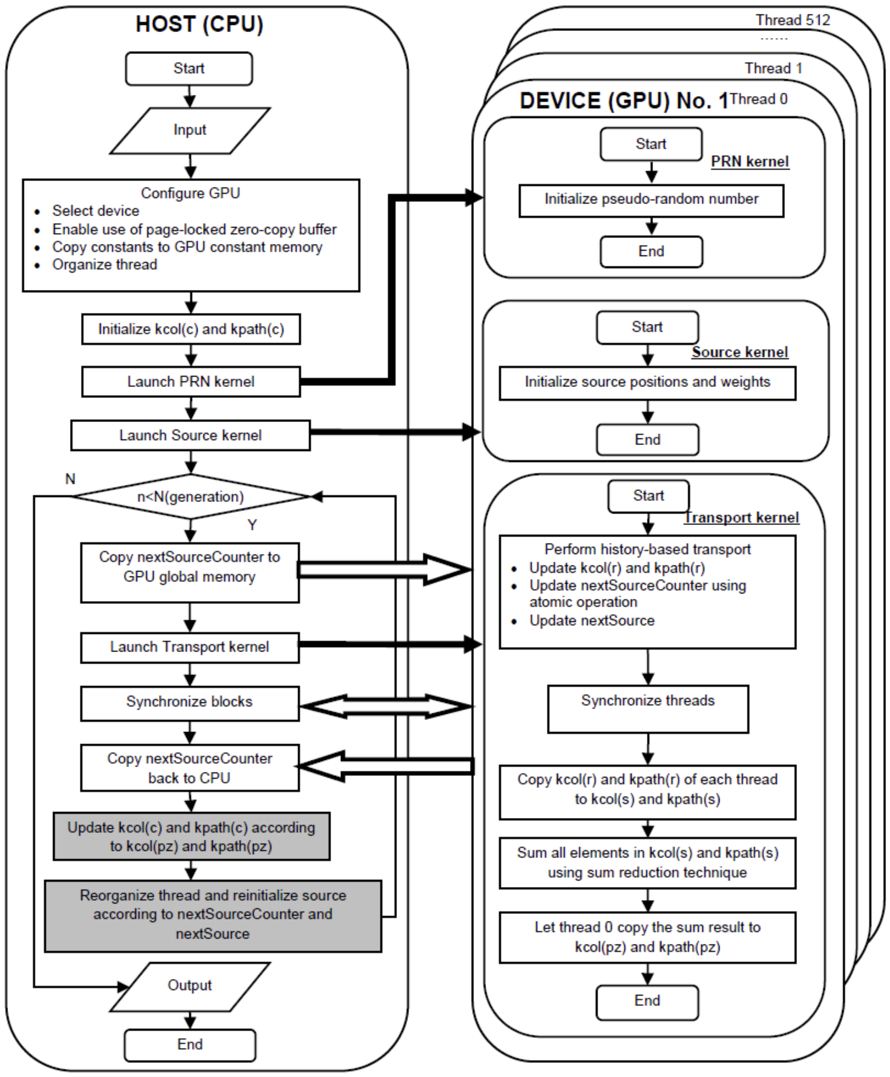
HOST (CPU)
Start
Input
Configure GPU
Select device
Enable use of page-locked zero-copy buffer
Copy constants to GPU constant memory
Organize thread
Initialize kcol(c) and kpath(c)
Launch PRN kernel
Launch Source kernel
N
n<N(generation)
Y
Copy nextSourceCounter to GPU global memory
Launch Transport kernel
Synchronize blocks
Copy nextSourceCounter back to CPU
Update kcol(c) and kpath(c) according to kcol(pz) and kpath(pz)
Reorganize thread and reinitialize source according to nextSourceCounter and nextSource
Output
End
Thread 512
......
Thread 1
DEVICE (GPU) No. 1
Thread 0
Start
PRN kernel
Initialize pseudo-random number
End
Start
Source kernel
Initialize source positions and weights
End
Start
Transport kernel
Perform history-based transport
Update kcol(r) and kpath(r)
Update nextSourceCounter using atomic operation
Update nextSource
Synchronize threads
Copy kcol(r) and kpath(r) of each thread to kcol(s) and kpath(s)
Sum all elements in kcol(s) and kpath(s) using sum reduction technique
Let thread 0 copy the sum result to kcol(pz) and kpath(pz)
End

Figure 2 Flowchart of the GPU-based eigenvalue calculation program. The "c" following kcol and kpath denote that the values are stored in the regular host memory. "pz" denotes the host page-locked zero-copy memory. "r" denotes the device register. "s" denotes the

device shared memory. For brevity, only one device is plotted. The boxes colored in grey are the serial code that cannot be parallelized.

2.1.2.1. Thread organization

The inner iteration in our code, i.e. the transport kernel was performed in parallel on the GPU. To obtain the device-level parallelism, the code was designed to automatically identify and choose the Fermi GPUs with compute capability above 2.0, and evenly dispense the histories of one certain generation to those GPUs. To obtain the thread-level parallelism, the histories assigned to each GPU was further evenly distributed to a large number of block threads. Specifically, the number of thread per block was set to be 512, and the number of block per grid was given by Equation 1 such that every history of the current generation was guaranteed to be assigned to one thread to simulate, with a few threads in a few blocks probably being idle, which was immaterial to the overall performance. The outer iteration, i.e. the generation loop was organized by CPU and was performed sequentially primarily because GPUs could not manage thread reorganization by themselves.

$$\#\text{ of blocks per grid} = \left\lfloor \frac{\#\text{ of histories of current generation} + \#\text{ of threads per block} - 1}{\#\text{ of threads per block}} \right\rfloor$$

Equation 1

2.1.2.2. Memory management

To maximize the performance, six different types of memories were utilized in the GPU-based MC code.

Page-locked zero-copy buffer: Initially, the CPU needed to schedule the transfer of intermediate data (such as the updated eigenvalues and the coordinates of neutrons generated from fission reaction) between memories on the CPU and GPU sides, resulting in heavy data traffic. To alleviate this problem, page-locked zero-copy buffer was used which was a special physical memory on the CPU side. It could be conveniently mapped to the address space of a GPU, and therefore be directly and rapidly accessed by the GPU program. Consequently, the intermediate data were not placed in the GPU memory, but directly sent to this buffer for storage, and CPU did not have to explicitly schedule any data transfer. The collision and path-length estimators stored in this memory are represented by kcol(pz) and kpath(pz) in Figure 2.

Per-thread Local memory: Aside from fission reaction, neutrons could also be created artificially by the splitting technique, in which the neutron with a large weight was decomposed into multiple neutrons with smaller weights. One of these neutrons would continue to be transported, while the rest of them were put in the per-thread local memory.

Global memory: A variable called nextSourceCounter was placed in this memory. It was used to record how many fission neutrons had already been generated in simulation of the current generation and was visible (i.e. modifiable) to all the threads in the current grid. A potential pitfall called “race condition” is very likely to occur when two threads simultaneously attempt to perform read-and-modify operations to that variable. The two threads first read the variable and get the same value, then perform their individual operations on it. For example, the first thread adds 2 to the value while the second thread adds 3. After that the two threads compete to see which one is the last to write its result to the variable. As a consequence, the value of the variable is not increased by 5, but only by 2 or 3, depending on which thread finally wins. To avoid this problem, atomic operation provided by CUDA was used to ensure that nextSourceCounter was always correctly updated by two or more racing threads.

Per-thread register: This memory was used to store all the per-thread variables, including the collision and path-length estimators calculated by each thread, represented by kcol(r) and kpath(r) in Figure 2.

Per-block shared memory: This memory was used to rapidly sum up the collision/path-length estimators over all the threads in a block. Sum reduction technique, which was reported to be 30 times faster than an unoptimized code [18] was adopted to effectively accelerate this process. The estimators residing in this memory are represented by kcol(s) and kpath(s) in Figure 2.

Constant memory: Constants such as cross-section data and geometry parameters were stored here.

The GPU code was written in CUDA C 4.1 and tested on a dual Tesla M2090 GPU system.

2.2. Hardware Implementation

The dual GPU system used in this paper contains two NVIDIA Tesla M2090 cards that are based on the latest NVIDIA Fermi architecture. Each GPU has 512 streaming processors (SP) and 6-GB memory. The double-precision peak performance is 665 GFLOPS and the memory bandwidth is 177GB/s [19].

3. RESULTS

The CPU and GPU programs were separately tested by running 10^6 initial histories and 1000 generations. The first 200 generations were made inactive. Double precision floating point was used throughout the code to guarantee the computation precision. The result was summarized in Table 1. We chose histories simulated per second as the speed indicator, which is equal to the total histories across all generations divided by the computation time.

Table 1 Comparison of CPU and GPU double-precision calculation speed for eigenvalue problem

Case	**Bare spherical core**		**Binary slab system**	
Histories per second on a CPU [s^{-1}]	4.69×10^5		1.57×10^5	
Histories per second on a single/dual GPU system [s^{-1}]	3.30×10^6	6.63×10^6	5.24×10^6	1.01×10^7
Speed-up factor of a single/dual GPU system	7.0	14.12	33.3	64.0

It is noticed that the speedup factors obtained under the bare spherical core and the binary slab geometry are very different, the former being much smaller than the latter. This is mainly a result of bandwidth limit: in the bare spherical core problem, each time a source neutron is generated, three double-precision floating point numbers representing x, y, z coordinates need to be copied to the page-locked zero-copy buffer, which will result in three times larger data throughput. Even so, use of this special buffer is still beneficial. The amount of data transferred from GPU to the page-locked memory and to the regular pageable memory was tested, and a bandwidth of 6262.7 MB/sec and 5478.4 MB/sec was detected respectively, indicating a gain of 10% in data transfer efficiency by using page-locked memory.

It is also observed that when two GPUs were used in tandem, the speedup factors were nearly doubled. However, this does not indicate a linear relationship between the speed gains and the amount of GPUs. The device-level parallelism is limited by the fraction of serial code, which in our case is part of the outer iteration that has to be executed sequentially, as is colored in grey in Figure 2. The limit on speedups can be quantitatively evaluated by Amdahl's law, expressed in Equation 2, where s is the serial fraction of the program (correspondingly, 1-s is the parallelizable fraction), and n is the number of processors, referring to the number of GPUs in the context of multi-GPU programming. As n goes to infinity, the speedup factors approaches 1/s.

$$\mathrm{Speedup}(s,n) = \frac{1}{s + \frac{1-s}{n}}$$

Equation 2

The maximum achievable speedup for the binary slab system eigenvalue calculation was evaluated. Time spent on the serial code and on the entire outer iteration was accurately measured. The result showed that the serial code occupied 1.65% of the total execution time, implying a maximum device-level speedup of 60.5, and equivalently a maximum total speedup of 2016 compared to the code running on one CPU.

4. DISCUSSION

Previous research adopted multiple methods to increase the performance of GPU-based MC code. One was to use the intrinsic transcendental function executed by the internal Special Function Units (SFU) on GPUs. The other was to use the single-precision floating point calculation to immediately obtain twice more FLOPS than use of the double-precision calculation on GPUs with Fermi architecture. However, both of them trade accuracy for speed and should not be encouraged in MC calculation. One needs to realize that the major performance limiter is the inconsistency between GPU execution mechanism and the history-based MC algorithm. On the GPU, threads are split into groups called warps. Each warp contains 32 threads and executes one common instruction concurrently. Highest performance will be achieved if all the 32 threads of a warp agree on their execution path. [20] If threads diverge at a certain data-dependent conditional branch, the warp will sequentially execute each branch. This phenomenon is called thread divergence. Because MC contains a large amount of conditional branches and which branch for a thread to enter is randomly selected, thread divergence becomes an inevitable problem affecting the overall performance. One possible solution is to revisit the vectorized event-based MC method developed in the 1980s that was intended to be implemented on vector supercomputers [21,22,23]. The advantage of this method is that at any time during the simulation, all the particles will be in the same event iteration, such as the free-flight analysis, collision analysis and boundary analysis, meaning that the iterations does not contain conditional branches any more. Since thread divergence can potentially be eliminated this way, one will be very likely to get a good performance gain.

5. CONCLUSION

A GPU-based MC code for eigenvalue calculation was developed. Two geometry configurations were simulated: a bare spherical core and a binary slab system. Double-precision floating point was used in the entire calculation to guarantee high accuracy. A total of 1000 generations of neutrons were simulated with the initial generation having 10^6 histories. The parallelism was achieved on both the device level and thread level. First, the total simulation task was evenly distributed to two NVIDIA Tesla M2090 GPUs. Second, the task was further divided on each GPU into many blocks each having 512 threads. The result showed that when a single GPU was used, the GPU code was 7.0 and 33.3 times faster than the CPU code for the two geometry configurations. These speedup factors became 14.12 and 64.0 when both GPUs were used. For the simulation using binary slab geometry, the upper limit of the speedups was found to be 2016. Vectorized MC method can be a candidate to mitigate thread divergence in MC calculation and further the per GPU speedup.

REFERENCES

1. K. Smith, “Reactor Core Methods,” Invited lecture at the *M&C 2003 International Conference*, Gatlinburg, TN, USA, April 6-10 (2003).
2. W. R. Martin, “Advances in Monte Carlo Methods for Global Reactor Analysis,” Invited lecture at the *M&C 2007 International Conference*, Monterey, CA, USA, April 15-19 (2007).
3. J. E. Hoogenboom, W. R. Martin, “A Proposal For A Benchmark To Monitor The Performance Of Detailed Monte Carlo Calculation Of Power Densities In A Full Size Reactor

Core," *International Conference on Mathematics, Computational Methods & Reactor Physics (M&C 2009)*, Saratoga Springs, New York, May 3-7 (2009).

4. J. E. Hoogenboom, W. R. Martin, B. Petrovic, "Monte Carlo Performance Benchmark For Detailed Power Density Calculation In A Full Size Reactor Core, Benchmark specifications, Revision 1.1, June 2010" http://www.oecd-nea.org/dbprog/documents/MonteCarlobenchmarkguideline_004.pdf (2010).
5. Workshop on high performance computing for industry: providing a competitive advantage to industry through high-performance computing accomplishments and a path forward, October 26-28, 2011, Rensselaer Polytechnic Institute. http://www.rpi.edu/hpcw/program.html (2011).
6. F. B. Brown, "Recent Advances and Future Prospects for Monte Carlo," *Progress in nuclear science and technology*, Tokyo, Japan, October 17-21, 2010, **Volume 2**, pp.1-4 (2011).
7. A. Heimlich, A. C. A. Mol, C. M. N. A. Pereira, "GPU-Based High Performance Monte Carlo Simulation in Neutron Transport," *2009 International Nuclear Atlantic Conference*, Rio de Janeiro, RJ, Brazil, September 27-October 2, 2009 (2009).
8. 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).
9. 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).
10. J. Tickner, "Monte Carlo simulation of X-ray and gamma-ray photon transport on a graphics-processing unit," *Computer Physics Communications*, **Volume 181**, Issue 11, pp.1821-1832 (2010).
11. X. Jia, X. Gu, J. Sempau, D. Choi, A. Majumdar, S. B. Jiang, "Development of a GPU-based Monte Carlo dose calculation code for coupled electron–photon transport," *Phys. Med. Biol.* **Volume 55,** pp.3077–3086 (2010).
12. P. P. Yepes, D. Mirkovic, P.J. Taddei, "A GPU implementation of a track-repeating algorithm for proton radiotherapy dose calculations," *Phys. Med. Biol*. **Volume 55,** pp.7107–7120 (2010).
13. A. Badal, A. Badano, "Fast and accurate estimation of organ doses in medical imaging using a GPU-accelerated Monte Carlo simulation code," *2011 Joint AAPM/COMP Meeting*, Vancouver, July 31-August 4, 2011 (2011).
14. http://www.oecd-nea.org/tools/abstract/detail/nea-1525 (2012).
15. S. Hissoinya, B. Ozell, "GPUMCD: A new GPU-oriented Monte Carlo dose calculation platform," *Med. Phys.*, Volume 38, Issue 2, pp. 754-764 (2011).
16. http://irs.inms.nrc.ca/software/egsnrc/ (2012).
17. X. Jia, H. Yan, X. Gu, S. B. Jiang, "Fast Monte Carlo simulation for patient-specific CT/CBCT imaging dose calculation," *Phys. Med. Biol.*, **Volume 57**, pp. 577–590 (2012).
18. http://developer.nvidia.com/cuda-cc-sdk-code-samples (2012).
19. "TESLA M-Class GPU Computing Modules Accelerating Science," http://www.nvidia.com/docs/IO/105880/DS-Tesla-M-Class-Aug11.pdf (2012).
20. "NVIDIA CUDA C Programming Guide V4.1," http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf (2012).

21. F. B. Brown, “Vectorized Monte Carlo, PhD Dissertation,” Department of Nuclear Engineering, University of Michigan (1981).
22. F. B. Brown, W. R. Martin, “Monte Carlo Methods for Vector Computers,” *Progress in Nuclear Energy*, **Volume 14**, Issue 3, pp. 269-299 (1984).
23. W. R. Martin, F. B. Brown, “Present Status of Vectorized Monte Carlo for Particle Transport Analysis,” *International Journal of Supercomputer Applications*, **Volume 1**, Issue 2, pp. 11-32 (1987).