Annals of Nuclear Energy 113 (2018) 506–518

Contents lists available at ScienceDirect

Annals of Nuclear Energy

journal homepage: www.elsevier.com/locate/anucene

Multigroup Monte Carlo on GPUs: Comparison of history- and event-based algorithms ☆

Steven P. Hamilton *,[1], Stuart R. Slattery [2], Thomas M. Evans [1]

Oak Ridge National Laboratory, 1 Bethel Valley Rd., Oak Ridge, TN 37831 USA

ARTICLE INFO

Article history:
Received 9 August 2017
Received in revised form 9 November 2017
Accepted 17 November 2017

Keywords:
Radiation transport
Monte Carlo
GPU

ABSTRACT

This paper presents an investigation of the performance of different multigroup Monte Carlo transport algorithms on GPUs with a discussion of both history-based and event-based approaches. Several algorithmic improvements are introduced for both approaches. By modifying the history-based algorithm that is traditionally favored in CPU-based MC codes to occasionally filter out dead particles to reduce thread divergence, performance exceeds that of either the pure history-based or event-based approaches. The impacts of several algorithmic choices are discussed, including performance studies on Kepler and Pascal generation NVIDIA GPUs for fixed source and eigenvalue calculations. Single-device performance equivalent to 20–40 CPU cores on the K40 GPU and 60–80 CPU cores on the P100 GPU is achieved. In addition, nearly perfect multi-device parallel weak scaling is demonstrated on more than 16,000 nodes of the Titan supercomputer.

1. Introduction

Effective design and analysis of many nuclear systems rely on the ability to accurately solve the radiation transport equation. Monte Carlo (MC) methods offer the most accurate radiation transport solutions, but they are subject to uncertainty due to persistent stochastic noise as a result of random sampling. Reducing this noise can be accomplished by increasing the number of simulated particle histories, but achieving the precision required for many applications comes at a substantial computational cost.

Recent trends in high performance computing favor vectorized, single-instruction multiple-data (SIMD) or single-instruction multiple-thread (SIMT) architectures such as GPUs or the Intel Xeon Phi processors. These processors offer performance at a significantly lower energy cost per floating point operation than traditional CPUs. The challenge of performing MC transport on a vectorized computing architecture is not new: the vector supercomputers that flourished in the 1980's led to the introduction of event-based MC (Brown and Martin, 1984). In a traditional MC algorithm (referred to in this paper as "history-based"), individual particle histories are simulated from the time they are created until their termination. On the other hand, the event-based approach processes groups of particles in batches based on the next event that the particles will undergo. Thus, a collection of particles undergoing geometric tracking will be processed together as a group, and particles scheduled to collide will be processed together. Processing particles together in this fashion allows the algorithm to exploit the vectorization capabilities of the computing architecture. Most recent work to adapt MC transport to GPUs has focused on event-based algorithms (Bergmann and Vujić, 2015; Xu, 2015; Ozog et al., 2015). Modern architectures, however, are far more versatile than the vector computers of decades past. While reducing thread divergence is still an important consideration for many GPU algorithms, it is not clear that such a drastic change from traditional CPU algorithms is necessary. Some evidence suggests that low-level thread divergence resulting from short-lived branching may not be detrimental to performance if the impact on memory bandwidth is taken into account (Scudiero, 2014).

This paper considers the solution of the multigroup form of the transport equation on GPUs. While many applications are focused only on continuous-energy transport, use of the simpler multigroup physics allows for algorithmic details to be investigated more thoroughly. In addition, while many performance issues will

☆ This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

* Corresponding author.
E-mail addresses: hamiltonsp@ornl.gov (S.P. Hamilton), slatterysr@ornl.gov (S.R. Slattery), evanstm@ornl.gov (T.M. Evans).
[1] HPC Methods and Applications Team, Reactor and Nuclear Systems Division.
[2] Computational Engineering and Energy Sciences Group, Computational Sciences and Engineering Division.

https://doi.org/10.1016/j.anucene.2017.11.032
0306-4549/

be different in continuous-energy transport, there are many commonalities between the approaches that naturally extend from multigroup to continuous energy. Where possible, features likely to carry over to the continuous-energy problem are identified. Furthermore, the multigroup MC approach continues to be used in codes such as Shift (Pandya et al., 2016) and the KENO-VI module of the SCALE package (SCALE, 2011). In addition, methods such as the implicit MC approach for nonlinear radiative transfer are exclusively formulated using the multigroup (multifrequency) approach (Fleck and Cummings, 1971).

This paper provides a comparison of traditional history-based and event-based MC transport algorithms on GPUs. The implementation details of each method are thoroughly described, including several new algorithmic developments for each approach. The remainder of this paper is organized as follows. Section 2 provides background on the Profugus code used in this work, as well as the GPU architecture. Section 3 describes the traditional history-based transport algorithm and its implementation for GPUs, including ideas for reducing thread divergence. Section 4 describes the implementation of an event-based algorithm, with particular focus on approaches to enhance performance through improved sorting and source generation. Section 5 provides numerical results comparing features of the history-based and event-based formulations, including scaling studies on the Titan supercomputer at the Oak Ridge Leadership Computing Facility (Oak Ridge Leadership Computing Facility, 2016). Concluding remarks and ideas for future work are given in Section 6.

2. Profugus GPU implementation

The studies in this manuscript were performed using the Profugus MC code (Profugus). Profugus is an open-source multigroup MC solver developed at Oak Ridge National Laboratory. It is designed to mimic the algorithms and design of the Shift MC code (Pandya et al., 2016). It can solve both fixed source and criticality (k-eigenvalue) problems. While Shift is designed to be a production-level analysis tool, the primary purpose of Profugus is algorithmic research. Therefore, Profugus only implements a subset of the functionality in Shift. For example, Shift uses either multigroup or continuous-energy nuclear data, while Profugus only implements the multigroup approach. Shift supports a wide range of geometric capabilities, while Profugus is limited to either a Cartesian mesh or the reactor toolkit (RTK) geometry which is optimized for modeling of pressurized water reactors (Pandya et al., 2016). Finally, Profugus only offers a small number of tally options, specifically a total flux cell tally and tallies necessary for performing criticality calculations. Shift, on the other hand, provides a wide variety of tally options. In all calculations in this document, the standard variance reduction technique of implicit capture (also known as *absorption suppression* or *survival biasing*), combined with Russian roulette, is employed. The Russian roulette weight cutoff is set to 0.25, and the survival weight is set to 0.5—the same settings used in the Shift transport code (Pandya et al., 2016).

A brief overview of the NVIDIA GPU architecture is provided, along with corresponding software considerations of the CUDA programming language (CUDA C programming guide, 2015). NVIDIA GPUs contain several independent streaming multiprocessors. In this report, three GPUs will be considered: the K20X and K40 devices of the Kepler generation and the P100 of the newer Pascal generation. The K20X and K40 GPUs contain 14 and 15 independent multiprocessors, respectively, while the P100 contains 56. Each multiprocessor can execute numerous threads simultaneously. These threads are grouped into collections of thread blocks that are assigned to run concurrently on the same multiprocessor. Within a thread block, the threads are grouped into sets of 32 threads known as *warps* which constitute the vectorization unit of the GPU. The 32 threads in a warp execute instructions together in lockstep: any instruction that must be executed by any thread in a warp must be executed by every thread in the warp. When branching instructions are encountered, the entire warp will execute each branch that is taken by any thread within the warp. When different threads within a warp take different code branches, thread divergence occurs. During execution of branches for which a given thread is inactive, the thread is *predicated*—that is, it will still execute all instructions, but the results of these calculations will be discarded. Thus, branching statements do not impact the correctness of the results computed by individual threads, but they may have a significant impact on the performance of the code.

Several different types of memory are present on GPUs. Data allocated in global memory are accessible by all threads on all multiprocessors on the GPU. *Global memory* represents the largest component of GPU memory, but it is also subject to the highest latency and lowest bandwidth. Global memory accesses can take advantage of some data caching, although the cache structure is much less sophisticated than a typical CPU cache. *Shared memory* is visible to all threads within a single thread block. Each thread block has its own portion of shared memory, and data allocated to shared memory in one thread block is not visible from any other thread block. Shared memory has extremely low latency and high bandwidth but is very limited in size, typically 48 kB distributed among all of the thread blocks executing on a given multiprocessor. Texture memory, or texture fetching, does not represent a different memory location; rather, it points to alternate hardware for fetching data from global memory. Texture fetches can take advantage of a richer cache hierarchy than standard global memory accesses, but only read-only access can be achieved.

The GPU capabilities described in this paper were implemented using the CUDA programming language, which was introduced by NVIDIA to facilitate the use of GPUs for general-purpose programming. CUDA uses the same syntax as C/C++, with additional constructs specifically targeted at programming for the GPU (a Fortran-compatible version of CUDA is also available). Special functions designed to execute on the GPU are known as *kernels*, and the process of calling a kernel from the CPU to execute on the GPU is known as a *kernel launch*. Due to limitations in the architecture, a number of C++ features (e.g., inheritance) are not available or they incur a significant performance penalty (e.g., dynamic memory allocation) within on-device code. For this reason, to adapt an existing C/C++ code to run on the GPU, it is generally necessary to rewrite a significant portion of the code base. Some other programming models are available for writing software for NVIDIA GPUs, including OpenACC (The OpenACC application programming interface, 2015), Kokkos (Edwards et al., 2015), and RAJA (Hornung and Keasler, 2014). While these models often simplify the syntax of writing GPU code, they are generally implemented by converting the user's code into CUDA, so they do not remove any of CUDA's limitations. Furthermore, because these models are designed to offer interoperability between different computing architectures, some features of the CUDA language are not available. For this reason, the CUDA language was used directly in this GPU implementation.

Support for multiple GPUs is achieved using domain replication—geometry and cross section data are stored independently on each device. Communication between devices is achieved by assigning one MPI task per GPU, which enables the use of multiple devices on a single compute node as well as devices located on different compute nodes. For eigenvalue problems, a version of the parallel fission bank algorithm from Romano and Forget (2012) is used with slight modifications as described in Pandya et al. (2016).