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

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

[a] *Oak Ridge National Laboratory, 1 Bethel Valley Rd., Oak Ridge, TN 37831 U.S.A.*

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.

Keywords: radiation transport, Monte Carlo, GPU

*Corresponding Author

Email addresses: hamiltonsp@ornl.gov (Steven P. Hamilton), slatterysr@ornlgov (Stuart R. Slattery), evanstm@ornlgov (Thomas 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

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 [1]. 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 [2–4]. 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 [5].

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 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 [6] and the KENO-VI module of the SCALE package [7]. In addition, methods such as the implicit MC approach for nonlinear radiative transfer are exclusively formulated using the multigroup (multifrequency) approach [8].

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 [9]. 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 [10]. 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 [6]. 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 [6]. 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 [6].

A brief overview of the NVIDIA GPU architecture is provided, along with corresponding software considerations of the CUDA programming language [11]. 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 [12], Kokkos [13], and RAJA [14]. 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 [15] is used with slight modifications as described in Pandya et al. [6].

3. History-Based Algorithm

This section provides a description of a GPU implementation of a traditional history-based algorithm in which parallelism is achieved by assigning individual particle histories to unique computational threads. Each thread within a kernel launch will track a single particle for its entire lifetime (except as noted in Section 3.3). With the exception of the details described in Section 3.2, the following discussion applies to both fixed-source and criticality calculations.

3.1. Cell Tallies

At this writing, the only type of tally—other than the k-effective tally described in Section 3.2—is a total (energy-integrated) path-length flux tally within geometric cells. The list of cells for which a tally will be performed is specified by the user. To prevent the memory overhead associated with storing multiple copies of the tally field, a single copy of the tally is stored in global memory, and updates to the tally values are accomplished by performing an atomic update of the corresponding value. Prior to the release of the Pascal generation GPUs by NVIDIA in 2016, atomic updates of double precision floating point values were not natively available. The Pascal generation introduced native hardware capability for double precision atomics [16]. Because Profugus performs all operations using double precision arithmetic, it is necessary to implement atomic operations in software rather than hardware for all GPUs prior to the Pascal generation. Sample code for computing a software double precision atomic add is provided in Listing 1.

Listing 1: Software-based double precision atomic add in CUDA.

```
__device__ double inline atomic_add_double(
    volatile double * const dest, const double val)
{
  // Union of double precision with a 64 bit int
  union U
  {
    unsigned long long int i;
    double t;
  } assume, oldval, newval;

  oldval.t = *dest;

  // Perform update until value doesn't change
  do
  {
    assume.i = oldval.i;
    newval.t = assume.t + val;
    // Use CUDA builtin atomicCAS
    oldval.i = atomicCAS( (unsigned long long int*)dest,
                          assume.i, newval.i );
  } while (assume.i != oldval.i);

  return oldval.t;
}
```

Figure 1 shows the performance implications of the atomic update of the tallies on two different GPUs: the K40 and the P100. The problem being solved is a uniform 16 cm $\times$ 16 cm $\times$ 16 cm cube with reflecting boundaries on all sides. A single energy group is used with a total cross section of 1.0 cm^{-1} and a scattering cross section of 0.9 cm^{-1}. A source is located uniformly throughout the domain. The domain is divided into a varying number of uniformly sized mesh cells and a cell tally is applied to every cell. When the number of mesh cells is small compared to the number of threads active at a given point in time, many threads will simultaneously attempt to modify the same values, so the number of collisions on the atomic updates will be large. For the software atomics on both the K40 and the P100 GPUs, the increase in runtime due to atomic collisions becomes evident when the number of mesh cells drops below approximately 100 cells. Remarkably, with the hardware atomic on the P100, there is never a noticeable impact from the atomic updates, even when there is only a single mesh cell. At high cell

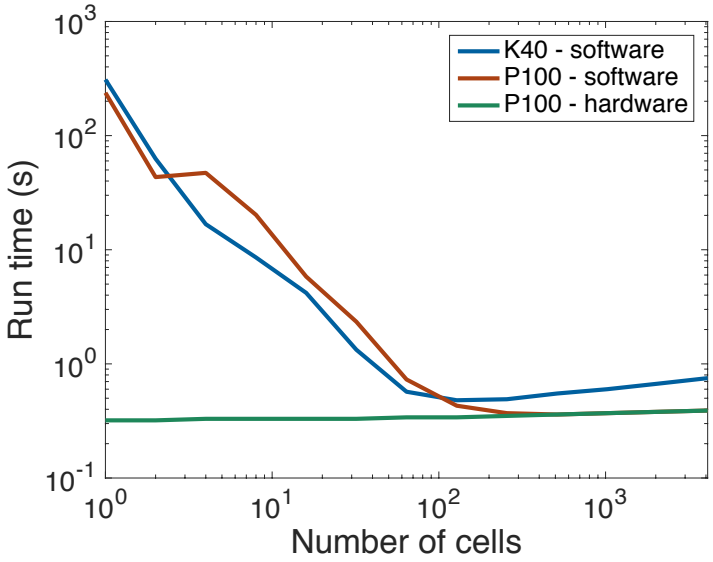


Figure 1: Run time as a function of cell count for uniform box

counts, the performance of both software and hardware atomics on the P100 are equivalent. This study indicates that when software atomics are required, it is preferable to avoid atomic updates where more than 1% of threads might simultaneously attempt to update the same value. If hardware atomic updates are available, it appears that the cost of atomic collisions is unlikely to ever be significant. Because almost any radiation transport problem of interest will have particle histories distributed across hundreds or thousands of spatial cells (even a single two-dimensional fuel assembly for a pressurized water reactor typically has on the order of 300 pin cells), the use of atomic updates for cell tallies is expected to offer acceptable performance.

3.2. Criticality Calculations

In an MC eigenvalue calculation, whenever a particle undergoes a collision, some number of fission sites are created at that location by sampling according to

$$n_f = \left[\frac{w}{k_\ell} \frac{\nu \sigma_f}{\sigma_t} \right], \tag{1}$$

where n_f is the number of fission sites created, k_ℓ is the eigenvalue estimate for cycle ℓ, w is the particle weight, and $[\cdot]$ represents an integer sampling operation. In the CPU algorithm, a vector of fission sites is dynamically

allocated during the random walk process by appending fission site information at each collision according to Eq. (1). This dynamic memory allocation on the GPU is prohibitively expensive, so an alternate approach is necessary. Storage for fission sites is pre-allocated by applying a factor of safety (typically 20%) to the current particle population. During the transport process, an integer is stored in global memory specifying the index of the next fission site. After each collision, if a nonzero number is sampled from Eq. (1), then that thread performs an atomic fetch/add on the fission site index and writes its fission site data into the retrieved site index. In this way, the retrieval of the site index is an atomic operation on an integer, while the writing of the fission site information can be performed non-atomically. Because an atomic update is performed on a single scalar quantity, this process may at first appear to be a performance concern. A simple argument, however, suggests that in practice it will not be problematic. The fission site sampling in Eq. (1) includes a factor of k_ℓ in the denominator to maintain the particle population at a constant value from cycle to cycle to within statistical fluctuation. This means that, on average, every particle history will only produce a single new fission site. In most nuclear systems, a particle will typically undergo tens or even hundreds of collisions before being killed. This is especially true when implicit capture is used for variance reduction [17]. Therefore, the vast majority of collisions will not result in any fission sites being generated. As a result, the number of atomic collisions while updating the fission site index is expected to be small, and no impact on performance is anticipated.

The other component of an eigenvalue calculation that requires attention is a tally to estimate the k_{eff} of the system. In this study, only a path-length tally is used. For a k_{eff} path-length tally, the tally quantity is a scalar value that must be updated every time a particle travels through a fissionable material. Because this is typically a very frequent occurrence, the use of an atomic operation is likely to be detrimental to performance. As an alternative, each thread is allowed to write into its own copy of the tally value during the transport kernel. A global reduction of the thread-local tally values is performed at the end of each cycle. However, the study on atomic performance in Section 3.1 suggests that on newer architectures supporting double precision atomic updates, it may be possible to perform even the k_{eff} tally using an atomic operation without loss of performance.

3.3. Mitigating Thread Divergence

One of the most significant features of NVIDIA GPUs is the single-instruction multiple-thread (SIMT) parallel thread model [11]. In this approach, groups of 32 threads, known as a warps, execute instructions together in lockstep. When instructions are encountered that involve conditional or branching logic (e.g., an "if" statement), all of the threads must execute every branch taken by any thread within the warp—a condition known as *thread divergence*. Threads are automatically marked as inactive during evaluation of branches that should be executed on a given thread. Therefore, the correct result will be produced regardless of the amount of branching in the instructions. However, a signficant performance penalty will result in such cases because many threads will effectively be sitting idle. Therefore, in order to improve performance, it is advantageous to limit the amount of thread divergence that can occur.

We begin the discussion of approaches for reducing thread divergence in a typical MC calculation with the introduction of pseudocode for a simplified version of a transport algorithm in Alg. 1. The algorithm consists of sequentially processing some specified number of particle histories. Each particle history progresses by alternating between predominantly geometric operations ("moveToCollision") and physics operations ("processCollision") until the particle history terminates. Because each particle history is independent, the loop over particle histories is a natural source of parallelism. In domain-replicated transport, parallelism is achieved by dividing the particle population across the available processors. In the GPU implementation, the loop is replaced by a CUDA kernel launch with a total number of threads equal to the number of particles. Although simplistic, this description of the algorithm illustrates the highest level location at which thread divergence can occur: particle histories will have different history lengths and therefore will execute the *while* loop a variable number of times.

Figure 2 shows the number of particles surviving as a function of the number of collisions experienced for 10^7 source particles in a fixed source version of the C5G7 MOX problem [18] in Profugus. The source term is uniformly distributed across the fuel region of the problem with a ^{235}U fission spectrum, approximating the behavior during a criticality calculation. After a small number of collisions (approximately 50), around 90% of the particles have disappeared, yet some particles survive for over 2000 collisions. This indicates that the GPU algorithm quickly reaches a point at which the vast

Algorithm 1 Basic transport algorithm

```
Get vector of source particles
for each particle do
    while particle is alive do
        Move particle to collision site                ▷ See Alg. 2
        Process particle collision                     ▷ See Alg. 3
    end while
end for
```

Algorithm 2 Move particle to collision site

```
Sample mean-free-path distance to collision, τ
while particle not at collision site do
    Compute distance to geometric boundary, x_bdry
    Compute total cross section, σ
    Compute distance to collision, x_coll = τ/σ
    if x_bdry < x_coll then
        Move particle to geometric boundary
        τ ← τ − σ · x_bdry
    else
        Move particle to collision site
    end if
end while
```

Algorithm 3 Process particle collision

```
Sample fission site creation (criticality calculations only)
Sample to choose between scatter or absorption
if particle scatters then
    Sample new energy group from scattering data
    Sample new direction
else
    Kill particle
end if
```

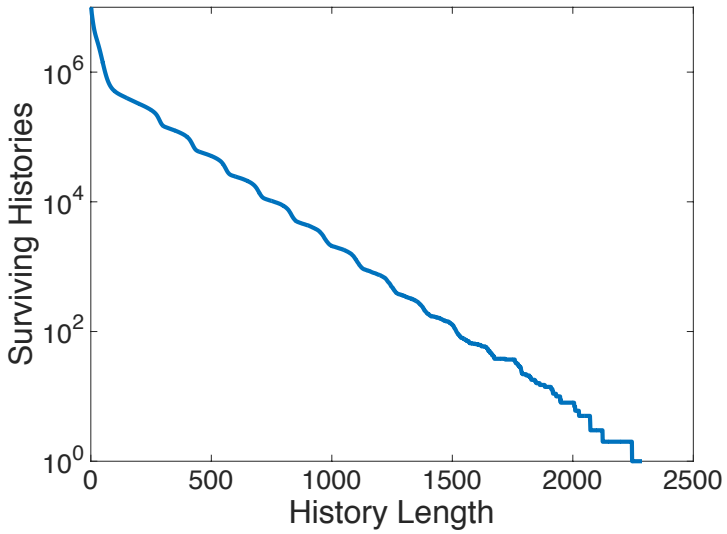


Figure 2: Number of particles surviving as a function of number of collisions.

majority of the threads on any given warp is inactive, leading to significant wasted resources.

Algorithm 4 offers a slight modification to Alg. 1, which attempts to address thread divergence due to variability in history length. Instead of allowing each history to continue indefinitely until termination, Alg. 4 applies a truncation criterion to limit each particle to a prescribed number of collisions. After the loop terminates, the population of particles is sorted, consolidating all surviving particles. This process is performed inside of an outer *while* loop and is continued until all particles are terminated. By periodically sorting the particles and consolidating those still alive, the algorithm provides the opportunity for warps where most threads were idle to be "restocked" with active particle histories. In practice, during the sort process the particle objects themselves are not relocated due to the computational cost of moving data [19]. Instead, a list of indices of active particles is maintained, providing a list of indices into the particle data. This makes the process similar to the reference remapping approach described in Bergmann and Vujić [2].

Figure 3 shows the runtime as a function of the sort frequency for Alg. 4 for three different energy group structures for a fixed-source version of the C5G7 problem [18]. This problem uses a Cartesian spatial mesh with a single mesh cell per fuel pin, with cross sections spatially homogenized over each pin

Algorithm 4 History-length truncation

```
Get vector of source particles
while any particles are alive do
    for each living particle do
        while particle is alive AND numCollisions < maxCollisions do
            Move particle to collision site                ▷ See Alg. 2
            Process particle collision                     ▷ See Alg. 3
        end while
    end for
    Sort/consolidate surviving particles
end while
```

using the XSProc module in SCALE [20]. The three energy group structures are all available in SCALE cross section libraries. A more detailed version of the C5G7 problem with increased spatial resolution will be presented in Sec. 5. At large intervals between sorting operations, particles are very rarely consolidated, making the algorithm essentially equivalent to the pure history-based algorithm of Alg. 1. The 8-group problem appears to be relatively insensitive to the sort interval, displaying only a minor reduction in run time as the interval between sorts is decreased. The 56-group problem behaves similar to the eight group problem at low sort intervals but shows a more significant increase in run time as the sort interval increases. The 252-group problem displays the greatest sensitivity to the sort interval: the run time decreases by a factor of 2.5 as the sort interval is decreased from the history-based limit to the optimal value at a sort interval of 10. For all three problems, the run time is quite flat when the sort interval is below 100. This indicates that there is significant benefit to occasionally sorting to account for dramatically varying history lengths, but there is little benefit to sorting very frequently.

In certain problem regimes, the modified approach described by Alg. 4 may not be effective. For example, in absorption-dominated problems, one particle may undergo only a small number of collisions before being absorbed or terminated by Russian roulette. In particular, if many of the geometry regions are optically thin, there may be significant variability in the number of geometric boundary crossings between collisions, resulting in variations in the number of iterations required to complete the *while* loop in Alg. 2. In this case, thread divergence occurs within a single movement to the next collision site; therefore, sorting again after a few collisions will have little

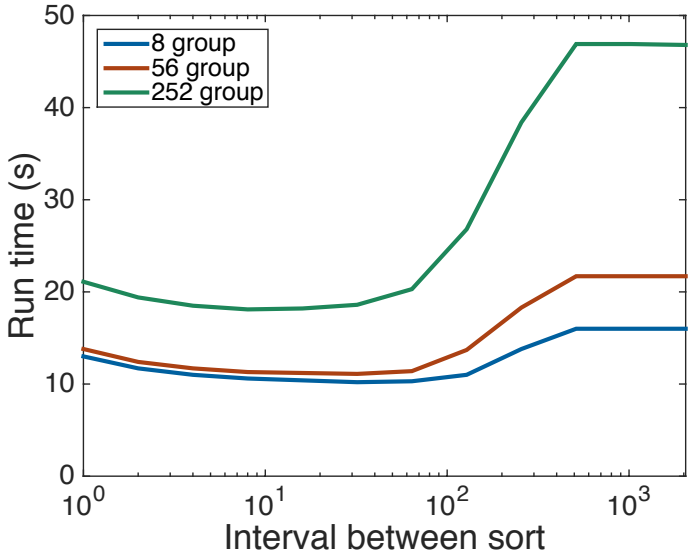


Figure 3: Number of particles surviving as a function of number of collisions.

impact on performance. In this regime, one approach may be to flatten the nested *while* loops into a single loop, thereby allowing the history truncation to address variations in either the number of geometric cells encountered or the number of collisions encountered. Note that this flattened loop structure is equivalent to the baseline history-based algorithm described in Bleile et al. [21]. In these investigations, little difference was observed between Alg. 4 and an equivalent algorithm employing a flattened loop structure for any of the problems considered.

Instead of (or in addition to) the process of consolidating active particle histories in Alg. 4, it would be possible to replace terminated particle histories with new particle histories from the source term. Such an approach will be discussed for an event-based algorithm in Sec. 4.1. This approach has not been implemented in the history-based algorithm for two reasons. First, the introduciton of new source particles into the particle vector complicates the static allocation of storage for fission sites in criticality problems described in Sec. 3.2. Second, in order to be effective, the number of particles in the vector would have to be much smaller than the number of source particles in order for there to be sufficient particles to restock the vector. However, reducing the size of the particle vector will reduce the overall efficiency of the

algorithm, offsetting the potential performance improvement.

3.4. Performance Optimizations

This section briefly describes a few algorithmic and compiler-related design decisions. The first choice concerns how data related to particle histories should be laid out in memory. In a standard CPU MC algorithm, it is common to have a data structure containing all information pertaining to an individual particle history. This information typically includes the particle's spatial position, direction, weight, energy (or energy group index), and a material identifier. It may also include additional information related to the geometric and/or physics state of the particle history; this may not be strictly necessary, but it may be advantageous to avoid computationally expensive recalculation of certain quantities. On a highly parallel computing platform with every computational thread assigned to a different particle history, it is necessary to have the data pertaining to a large number of particle histories allocated simultaneously. One approach for storing these data is to have a structure containing all data for a single particle and then to allocate an array of these structures with a length equal to the number of particles to be transported. This is known as an *array of structures* (AOS) approach. A different approach is to maintain separate arrays for each data component with an array of particle positions, an array of particle directions, etc. Each array would have a length equal to the number of particles. This is known as a *structure of arrays* (SOA) approach.

For computing architectures in which the threads are largely independent, such as multi-core CPUs, it is often recommended to use an AOS storage pattern. If multiple components of a given structure are accessed close to each other, then the AOS pattern allows for a high cache efficiency. Loading one component of the structure will cause many of the other components to be loaded into cache, causing subsequent accesses to data from the same structure to be already available in cache. In contrast, on computing architectures where threads (or at least groups of threads) operate in a collective fashion, an SOA approach is often recommended [22]. The SOA approach can improve performance by minimizing the loading of unused data from memory, thereby maximizing the available bandwidth for needed data. While the SOA approach is often strongly advised for GPU programming [22], recent studies show that an AOS approach may preferable at times, particularly for problems with challenging memory access patterns [23].

Another factor that can affect performance on GPUs is the availability of an L1 cache. On both Kepler and Pascal generation NVIDIA GPUs, all memory transactions use an L2 cache [16, 24], similar to conventional CPUs. However, the two architectures differ on the functionality of the L1 cache. On the Kepler architecture, the L1 cache is disabled by default and must be explicitly enabled via an argument to the compiler ("opt-in" caching). In addition to varying the cache behavior of the device, enabling the L1 cache on the Kepler architecture also changes the nature of memory accesses. With the L1 cache disabled, global memory transactions load a minimum of 32 bytes of data. When the L1 cache is enabled, the minimum amount of data loaded in a memory transaction increases to 128 bytes [24]. For some programs, increasing the minimum size of a memory transaction may result in an increase of memory traffic involving unused data, negatively impacting performance. The Pascal architecture significantly changes the behavior of the L1 cache. The most notable difference is that the L1 cache is enabled by default, but it can be disabled at compile-time. The other major difference is that the size of all memory transactions is 32 bytes, regardless of the L1 cache configuration. On both Kepler and Pascal devices, L1 cache and shared memory occupy the same physical memory. The programmer can choose to maximize the amount assigned to L1 cache, maximize the amount assigned to shared memory, or provide an even split between the two. In all of the studies using the L1 cache, the amount of L1 cache available has been maximized because no shared memory is being used.

Figures 4 and 5 show the performance of Profugus for the 252-group variant of the C5G7 benchmark problem for Kepler (K40) and Pascal (P100) GPUs, respectively. These figures show the variation in performance for various sort frequencies (as described in Section 3.3) resulting from changing the particle data layout between AOS and SOA, as well as changing the use of L1 cache. Figure 4 shows significant improvement in performance by using the L1 cache on the Kepler GPU, but only a slight advantage by changin from an AOS to an SOA storage approach. The overall shape of the curves as a function of sort frequency does not vary significantly between cases. Figure 5 shows that there is overall less sensitivity to caching or data layout on the Pascal device. There is a slight performance advantage to using an AOS data layout versus an SOA layout, but only a very minimal advantage to using the L1 cache. Overall, the decreased sensitivity to these parameters may indicate that newer GPUs will be better suited for problems with difficult memory access patterns such as MC particle transport.

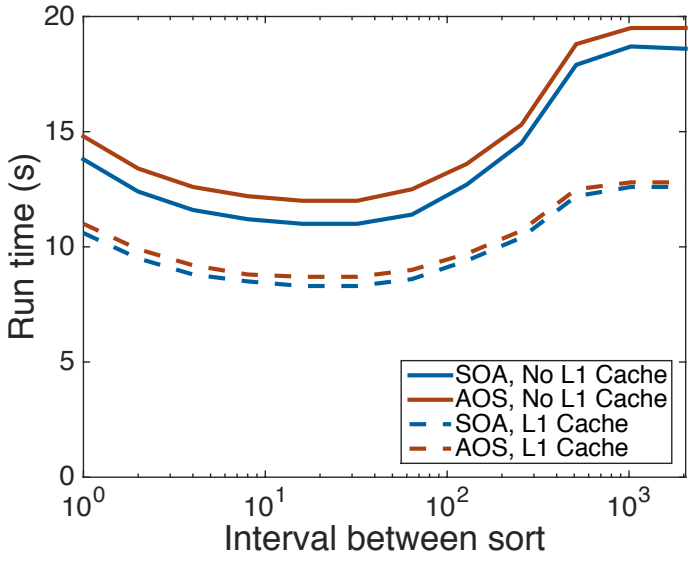


Figure 4: Performance as a function of sort frequency for K40 GPU

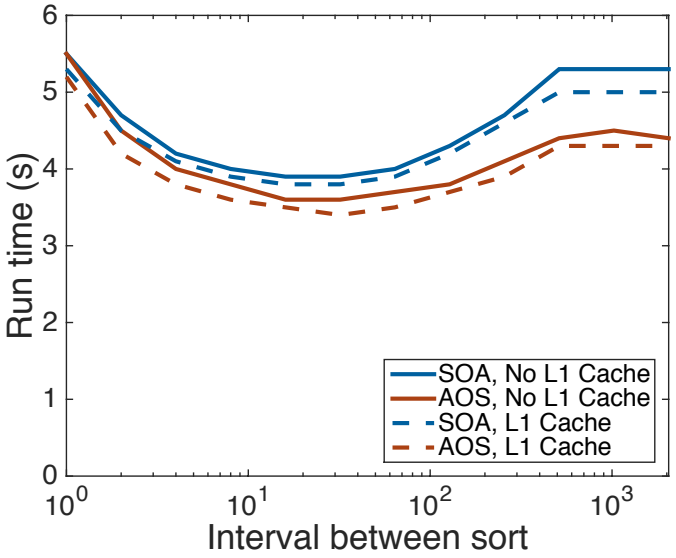


Figure 5: Performance as a function of sort frequency for P100 GPU

4. Event-Based Algorithm

Event-based transport algorithms are an alternative to the history-based transport algorithms described in Section 3. Event-based approaches were originally introduced to perform MC transport on the vector computers that dominated the high performance computing arena in the 1980s and 1990s [1]. Rather than following a single particle history from birth to death, in event-based transport, particle interactions are processed one event at a time. At each step, particles about to undergo the same action (e.g., compute the distance to the next geometric boundary) are processed together. By processing one event for a collection of particles, it is possible to take advantage of the vectorization capability of the processor. A significant number of studies concerning MC transport on GPU or Intel Xeon Phi architectures have focused on event-based approaches [2, 4, 21, 25].

Event-based algorithms offer several potential advantages for execution on GPUs. The obvious first advantage is that most branching logic will be handled outside the execution of GPU kernels, resulting in less thread divergence within kernels and therefore an improved utilization of vector arithmetic units. Because MC transport is more likely to be dominated by memory latency and/or bandwidth rather than arithmetic operations, it is not clear that improved arithmetic performance alone can result in improved overall performance. Another potential advantage of an event-based approach is that it allows for the creation of multiple specialized kernels of smaller complexity than the monolithic kernel required for a history-based approach. The reason this is significant is that more complex kernels typically require the use of a large number of GPU registers (thread-local variables). Because a finite amount of register space is available on each GPU multiprocessor, using a large number of registers can ultimately limit the number of threads that can be active simultaneously. The specialized kernels of an event-based method may be able to achieve a higher occupancy (the ratio of the number of active threads in a kernel to the maximum possible active threads) than a corresponding history-based method. While achieving high performance is possible even at low occupancy [26], increasing occupancy is typically desirable [11].

Algorithm 5 provides a basic event-based transport algorithm. This algorithm is similar to the event-based method proposed in Bleile et al. [21]. The algorithm leaves unspecified the number of event types that are defined. In Ref. [21], three distinct events are used: collision processing, material interface

crossing, and cell boundary crossing. In the initial Profugus implementation, two event types are used: moving a particle to its next interaction (which could be to a collision site or to the next geometric boundary), and processing a collision. Changing of materials is handled as part of the movement to a geometric boundary and thus is not treated as a separate event type. The event-based algorithm for continuous-energy MC transport presented in Bergmann and Vujić [2] defines more than a dozen event types.

Algorithm 5 Event-based transport algorithm

```
Get vector of source particles
while any particles are alive do
    Identify event for each particle
    for each event type do
        Process all particles of this event type
    end for
end while
```

4.1. Source Event

In many circumstances, the number of particles to be transported exceeds the number of particles that can be allocated on the GPU at one time. A standard remedy is to break up the particle population into batches that fit into device memory and then to process each batch sequentially. However, this approach introduces a potential performance concern. The number of particles active within a batch decreases as particles are absorbed (killed by Russian roulette) or as they leak out of the system at a rate analogous to that shown in Fig. 2. Once the number of particles falls below the number of threads needed to keep the device occupied, the algorithm will operate at a reduced efficiency. When multiple batches are treated sequentially, this regime of reduced efficiency is encountered not just one time, but once for each batch of particles.

We propose a possible remedy for this situation by introducing a "source event" to the list of events in Alg. 5. A particle vector size is selected that will fit in the GPU memory and is initially populated with source particles. During each event processing cycle, locations in the particle vector corresponding to completed particle histories can be replaced with new particles from the source. In this way, the vector is continuously replenished with new particles, keeping the vector fully occupied for a larger portion of the calculation than

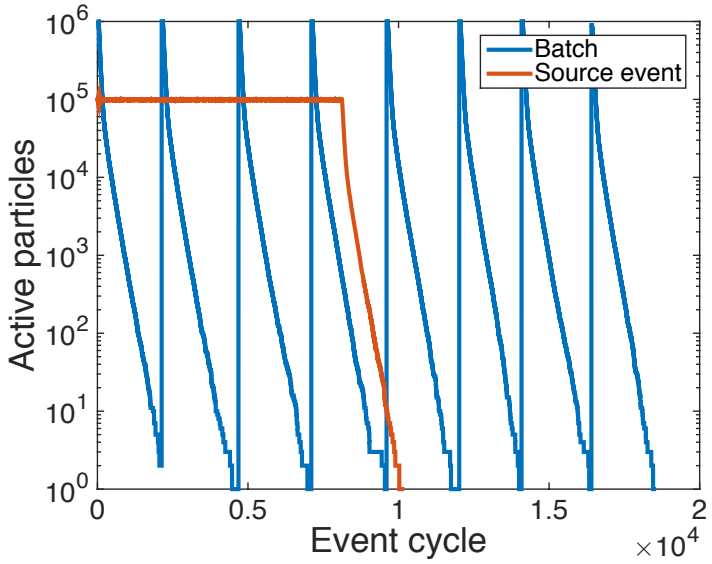


Figure 6: Active particle population as a function of event cycle for batch approach with a particle vector size of 10^6 and the source event approach with a vector size of 10^5.

with an ordinary batching approach. The regime of reduced efficiency is only encountered a single time at the conclusion of all particle histories rather than at the end of each individual batch. Figure 6 shows the number of active particles in the vector as a function of event cycle for the standard batching approach, as well as for the source event approach. Both approaches transport a total of 10^7 histories, but the batch approach uses a vector size of 10^6, and the source event approach uses a vector size of 10^5. Despite the smaller vector size, the source event method requires approximately half as many event cycles as the batch method due to the larger average number of active particles.

4.2. Event Determination

In Alg. 5, several possibilities exist for how to identify and select particles of a given event type. One approach is to simply launch each event kernel over the entire current vector of particles. Each kernel then selects particles with the correct event type to process. This approach causes each kernel launch to use many more threads than the number of particles to be processed, leading to a significant number of idle threads. This approach was considered in Bleile at al. [27] and deemed to be uncompetitive. A second approach is to

sort the entire vector of particle data according to the particle event type so that particle data corresponding to particles of each event type is stored contiguously. As discussed in Sec. 3.3, the cost associated with the data movement to sort the entire particle vector is typically prohibitively large. Bergmann and Vujić [2] reached the same conclusion for a continuous-energy event-based algorithm. An alternative approach is to maintain a list of indices of—or equivalently, pointers to—particles corresponding to each event type. This is the approach favored in previous work [2, 27], and it is the approach adopted in this study.

An algorithmic choice remains regarding how the list of indices for each event type should be generated. A simple choice is to form a list of particle event/index pairs and sort them by the event type. The indices are then permuted into groups corresponding to each event. This process can be accomplished using the Thrust [28] "sort_by_key" function, with the event types as the keys and the indices as the values. Although this method involves less data movement than sorting the entire vector of particle data, it still results in a large cost due to the sort. In Bleile at al. [21], the indices are determined by first performing an exclusive scan (also known as a prefix sum), followed by a scatter operation to generate lists of indices corresponding to each event type. This approach appears to involve only a small amount of data movement to complete. One final approach that has not previously appeared in the literature is to add the particle indices to lists of each event type from within each event kernel. At the conclusion of each event kernel, each thread performs an atomic add on a global integer containing the number of that event type recorded so far in the current event cycle. This process is very similar to the approach to store fission site data described in Sec. 3.2. This method can result in a large number of collisions on the atomic operation, but the results from Sec. 3.1 suggest that atomic operations may not be a barrier to performance, particularly on newer architectures. This primary advantage to this atomic-based method is that no processing of indices or data movement needs to occur between event cycles.

Figure 7 provides a comparison between the scan and scatter approach to computing indices versus the atomic update method for the 8-group and 252-group fixed-source version of the C5G7 problem described in Section 3 for varying particle vector sizes. The total number of particles transported in all cases is 10^7. The atomic-based method appears to be more algorithmically scalable, displaying significantly less sensitivity to the particle vector size than the scan and scatter method, particularly at small particle vector sizes. The

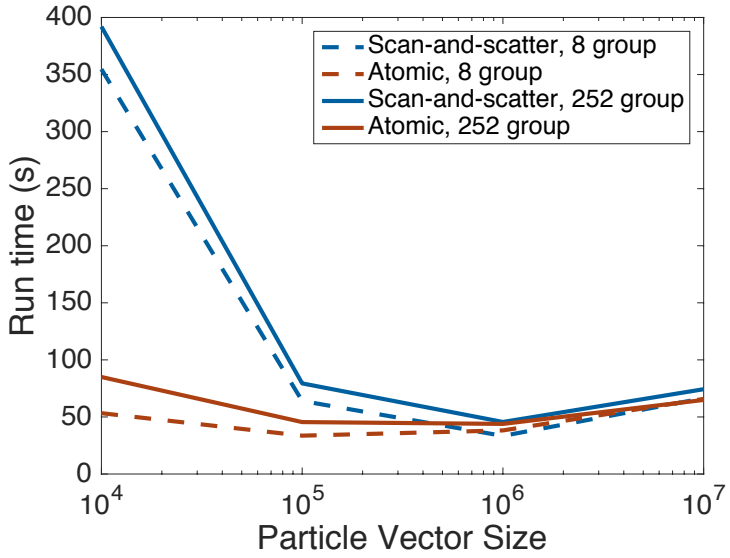


Figure 7: Comparison of index updating approaches in event-based GPU algorithm.

atomic approach achieves its best performance at a vector size of approximately 10^5, while the scan and scatter approach is optimal at a vector length of around 10^6. The performance at the optimal vector size is very similar for both methods. Achieving performance at low vector sizes may be important in some applications, particularly for eigenvalue calculations in which it may be desirable to run a large number of cycles with relatively few particles per cycle. Because the behavior is likely to have some problem-dependent sensitivity, the atomic-based approach is preferred due to its flatter overall performance profile.

5. Results

This section presents a comparison of the intra-node performance of the GPU implementations in Profugus to the corresponding CPU implementation, which uses only distributed-memory parallelism using MPI. Two computing architectures are considered. The first is a single-node machine containing two eight-core Intel Xeon E5-2630 v3 CPUs operating at 2.4 GHz along with four NVIDIA Tesla K40 GPUs. The second architecture is the IBM Power System S822LC for High-Performance Computing [29], which contains two ten-core IBM Power8+ processors and four NVIDIA Tesla P100 GPUs. In the following discussion, these machines will be abbreviated as *Xeon/K40* and *Power8/P100*, respectively.

For all problems described below, the history-based algorithm uses the history length truncation approach of Alg. 4 with a sort interval of 16. It uses an AOS data layout with the GPU L1 cache enabled. The event-based results use the source event method described in Sec. 4.1, as well as the atomic-based index updates described in Sec. 4.2.

5.1. Kobayashi Problem 3

The first test case is problem 3 from the Kobayashi benchmark problems [30]. This problem contains a 3D dog-leg void duct surrounded by a shield material with overall dimensions of 60 cm $\times$ 100 cm $\times$ 60 cm. The shield material is either a pure absorber (problem 3A) or a scattering material (problem 3B). The total cross section of the shield material is 0.1 cm^{-1}, and the scattering cross section in problem 3B is 0.05 cm^{-1}. The full geometric specification can be found in the original reference. The problem is modeled in Profugus using a uniform 1 cm Cartesian mesh.

Figure 8 shows the run times for Kobayashi problem 3A for a range of particle history counts. On both architectures, the CPU offers the lowest runtime at small particle counts because there is not enough parallel work to fully occupy the GPU. As the particle count is increased, the GPU is eventually saturated, and the history-based GPU algorithm outperforms the CPU. The event-based algorithm shows the same general trend as the history-based algorithm, but it is slower by a nearly constant factor for all cases. The results for Kobayashi problem 3B are very similar and are not shown. Tables 1 and 2 provide the run time and effective CPU core count for the simulation of problems 3A and 3B on both computing architectures. The effective CPU core count for a GPU simulation is defined relative to a CPU

Table 1: Run time and CPU core equivalent on Xeon/K40 architecture for Kobayashi problems with 10^8 histories.

	Problem 3A		Problem 3B	
Method	Run time (s)	Eff. cores	Run time (s)	Eff. cores
CPU (16 core)	18.0	-	46.6	-
GPU, event-based	66.1	4.4	162.4	4.6
GPU, history-based	9.2	31.3	35.2	21.2

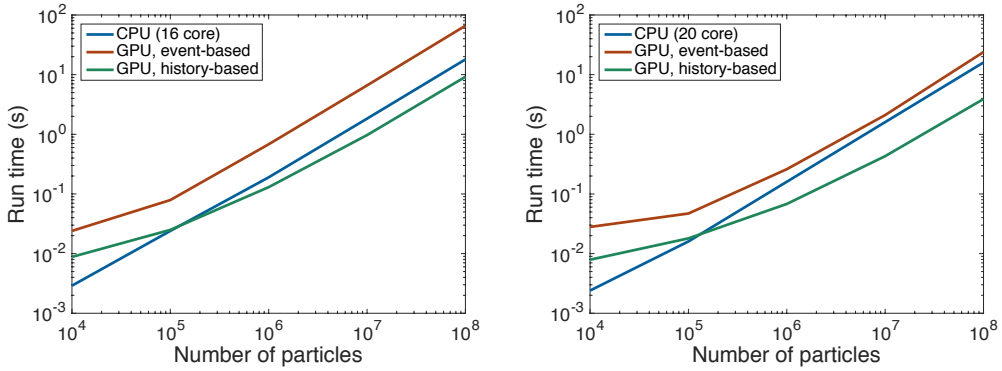
simulation as

$$\text{Effective CPU cores} = \frac{(\#\text{ CPU cores}) \times (\text{CPU run time})}{\text{GPU run time}}. \tag{2}$$

and provides a measure of the number of CPU cores that would be required to provide the same run time as the GPU simulation (assuming ideal parallel scaling with core count). The history-based algorithm significantly outperforms the event-based method in all cases, although the disparity is less severe on problem 3B than on 3A. On both architectures, the event-based algorithm offers less performance than the full CPU, while the history-based method outperforms the CPU, providing performance equivalent to 60–80 CPU cores on the Power8/P100 architecture. It is worth noting that the performance *per core* is nearly identical on the older Xeon processor and the newer Power8 processor and the total processor performance is only slightly improved on the Power8. On the other hand, the performance improves by around a factor of three when moving from the K40 to the P100 GPU. This appears to indicate that architectural improvements are currently progressing more rapidly on GPUs than on CPUs. We note that the CPU implementation is not exploiting the vector capabilities that are available on most modern CPUs and therefore some performance improvement is possible by utilizing the vector unit. However, considering studies aimed at exploiting the vector units on CPUs for MC transport have generally found limited success [31], we believe that the comparison presented here is as fair as can reasonably be achieved.

5.2. 3D C5G7 Problem

The next test problem is the well-known C5G7 MOX benchmark problem introduced in Sec. 3. Instead of the more common problem definition which



(a) Xeon/K40 **(b)** Power8/P100

Figure 8: Performance of algorithms for Kobayashi problem 3A.

Table 2: Run time and CPU core equivalent on Power8/P100 architecture for Kobayashi problems with 10^8 histories.

	Problem 3A		Problem 3B	
Method	Run time (s)	Eff. cores	Run time (s)	Eff. cores
CPU (20 core)	16.0	-	38.0	-
GPU, event-based	23.9	13.4	53.1	14.3
GPU, history-based	3.9	82.1	12.5	60.8

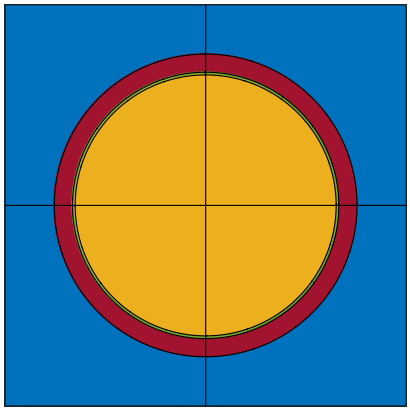


Figure 9: Fuel pin representation in Profugus for C5G7 problem.

has a total axial height of 64.26 cm [18], the earlier problem definition which has a total height of 214.2 cm [32] is considered. Two variants of the problem of varying complexity are considered. The first variant uses a Cartesian spatial mesh with a single mesh element for each pin cell in the problem. Instead of the seven-group cross sections provided in the benchmark specification, eight-group cross sections are used which are spatially homogenized using the SCALE package [20]. The second problem variant uses the reactor-based RTK geometry described earlier. To provide more detail than the benchmark specification (which only contains two materials per pin cell), the material and geometric specifications that were used to define the C5G7 materials are used [33]. These specifications contain four materials (fuel, gap, clad, and moderator) for each pin cell. Each pin is further divided with segments along the x and y midlines within each pin as shown in Fig. 9, providing 16 geometric regions within each pin cell. Pin-resolved 252-group cross sections were generated, again with the SCALE package. In each of the following cases, 10^7 histories per cycle are simulated, with 50 inactive and 100 active active cycles.

Tables 3–4 show the run time and effective CPU core count for each algorithm for the pin-homogenized and pin-resolved problems, respectively. As observed in Sec. 5.1, the performance per core between the Xeon and Power8 CPUs is nearly identical, while a dramatic performance improvement is seen when moving from the K40 to the P100 GPU. The history-based algorithm significantly outperforms the CPU and the event-based algorithm on both architectures, achieving equivalent performance to 20–40 CPU cores

Table 3: Run time and CPU core equivalent for 8-group pin-homogenized 3D C5G7 problem. The Xeon CPU results use 16 cores, and the Power8 CPU results use 20 cores.

	Xeon/K40		Power8/P100	
Method	Run time (s)	Eff. cores	Run time (s)	Eff. cores
CPU	3067	-	2457	-
GPU, event-based	6360	7.7	2857	13.8
GPU, history-based	1228	39.9	528	74.5

Table 4: Run time and CPU core equivalent for 252-group pin-resolved 3D C5G7 problem. The Xeon CPU results use 16 cores, and the Power8 CPU results use 20 cores.

	Xeon/K40		Power8/P100	
Method	Run time (s)	Eff. cores	Run time (s)	Eff. cores
CPU	8564	-	9448	-
GPU, event-based	14 312	9.6	5667	33.3
GPU, history-based	5894	23.2	2350	80.4

on the K40 and 75–80 CPU cores on the newer P100. The event-based algorithm, on the other hand, only achieves a maximum effective core count of around 10 cores on the the K40 and 33 cores on the P100. While the history-based method offers about the same speedup on the P100 for both problems, the event-based approach shows a much greater speedup on the more complex pin-resolved problem.

5.3. Discussion of Intra-Node Results

The superior performance of the history-based method relative to the event-based approach on the GPU appears counter to the expectations of many researchers, considering the emphasis on event-based methods to date in the literature. Although the precise reason for the difference in performance is likely due to a variety of factors, we speculate that the performance difference between the methods is predominantly due to two competing effects: idle threads due to branch divergence, and efficiency of memory accesses. When multiple threads within a GPU warp access data from nearby memory locations, the requests can be consolidated into a reduced number of memory transactions and memory coalescing is said to occur. In the limit when

the threads in a warp all access contiguous memory locations, all requested data can be provided by a single memory transaction, greatly improving efficiency. In the history-based algorithm, all data pertaining to individual particle histories are stored contiguously. Because the neighboring threads are assigned to neighboring particles, accessing particle data in the history-based approach results in nearly fully coalesced memory accesses. In the event-based approach, however, threads are re-assigned to different particles at each event cycle. This prevents significant coalescing when accessing particle data and reduces overall performance. Thus, while the event-based method is much better at reducing thread divergence, it is much less efficient in its memory access patterns, which seems to have a greater impact on performance.

In order to demonstrate the detrimental impact of non-coalesced memory accesses, Fig. 10 shows the behavior of the modified history-based algorithm for the same problem as Fig. 4 on the K40 GPU for a case where the assignment of particle indices to threads is modified by either striding or randomizing the indices. For the strided cases, each thread is assigned a particle index the differs from the previous thread by the stride (e.g., for the stride 2 case, thread 1 is given particle 1, thread 2 is given particle 3, etc.). As the stride is increased, the amount of coalescing of particle data is reduced, leading to diminished performance. When the particle indices are completely randomized, the performance is even further degraded because essentially no memory coalescing occurs. The event-based algorithm leads to more disjoint memory accesses due to processing different event types, thus reducing the performance that can be achieved.

Another issue working against the event-based approach is the inability to exploit cache and GPU-dependent memory spaces. The short-lived kernels in the event-based method offer little opportunity for reuse of data within a kernel. On the GPU, the entire cache is invalidated in between kernel launches. When values are only used once during a given kernel, which is typical in the event-based method, there is no opportunity to take advantage of performance improvements due to caching. Similarly, NVIDIA GPUs offer several unique memory features, including shared memory and texture fetching. Shared memory acts as a very low latency, programmable cache that is shared among the threads in a thread block. As with the standard cache, this space is flushed after every kernel launch, making its use in short-lived kernels difficult. Texture fetching offers a mechanism to exploit data locality by using a more sophisticated caching procedure than standard memory accesses. As with standard cache and shared memory, texture fetches rely

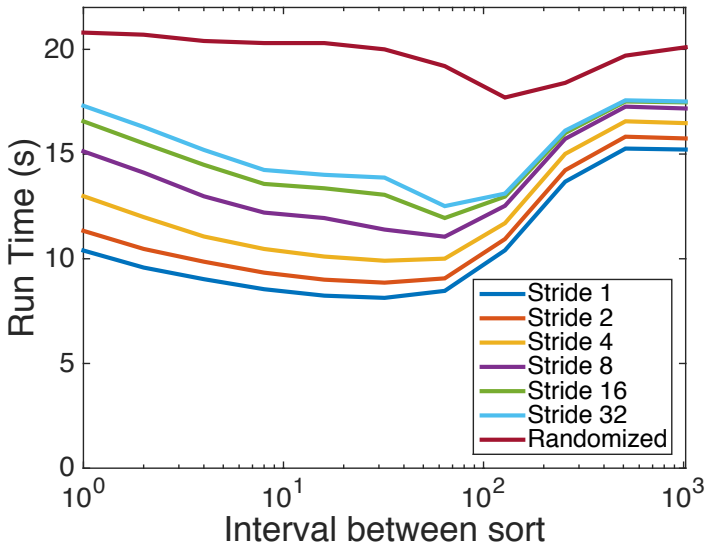


Figure 10: Run time as a function of node count for C5G7 problem on Titan.

on data reuse and would therefore not be easily exploited in the event-based algorithm. While the history-based approach described in this paper does not use shared memory or texturing fetching, future algorithmic improvements could exploit these features.

5.4. Distributed-Memory Scaling

So far, this paper has focused on the ability of Profugus to use the GPU on a single node. This section briefly illustrates the ability of the algorithms to use multiple GPUs. As noted in Sec. 2, inter-node parallelism in Profugus is achieved using domain replication with rebalancing of fission sites in eigenvalue calculations using a parallel fission bank algorithm. This section presents a parallel scaling study for the pin-resolved C5G7 problem described in the previous section on the Titan supercomputer [9] at the Oak Ridge Leadership Computing Facility. Titan is a Cray XK7 computer with a peak theoretical performance of more than 20 petaflops. It contains 18,688 compute nodes, each containing a 16-core 2.2 GHz AMD Opteron 6274 CPU and a single NVIDIA K20X GPU. In this study, only 10 eigenvalue cycles were performed. While this is an insufficient number of cycles to converge the fission source or produce accurate tally results, it is suitable for assessing parallel performance because the performance of each eigenvalue cycle is nearly identical. This

study is conducted as a weak scaling study, with the number of particles per node fixed at 10^7.

Figure 11 shows the run time for the CPU algorithm (16 cores per node), the event-based GPU method, and the history-based GPU method as the function of the number of compute nodes. Figure 12 shows the same data, with each method normalized to its value at one node to produce a weak scaling efficiency curve. As expected from the results in the previous section, the history-based method yields a lower run time than the other methods. In terms of parallel scaling, the CPU algorithm performs quite well, achieving greater than 80% weak scaling efficiency on nearly the entire Titan machine. Some degradation in performance is observed, however, particularly at the largest node counts. Even better parallel scaling effiency ($> 97\%$ on approximately 150,000 CPU cores) has been observed for continuous-energy calculations in Shift [6], the production MC code upon which Profugus is based. The reason for the higher efficiency in Shift is that the increased computational cost of continuous-energy calculations more effectively hides the time spent in communication. Remarkably, despite having a smaller run time than the CPU approach, the history-based algorithm shows no loss of efficiency over the entire range of node counts. The reason for the difference in behavior between the CPU and GPU is that the GPU is only using a single MPI task per node, while the CPU is using 16 tasks per node. At every data point, the CPU therefore requires a factor of 16 more MPI tasks, making the communication costs comparatively more expensive and thus reducing parallel efficiency. Note that it is expected that this behavior would be different for a CPU-based code using a hybrid MPI/threading approach, where the number of MPI tasks per node could be limited to a small number. For such a code, it is expected that the parallel performance would be similar to that shown by the GPU approaches. The event-based algorithm shows scaling behavior similar to that of the history-based algorithm except for a single jump in run time from 512 to 1024 nodes. The remainder of the scaling curve beyond 1024 nodes appears to be flat, just as with the history-based method. The reason for the jump at 1024 nodes is not clear, although it is presumed to be due to the load on the machine during the time when these runs were performed.

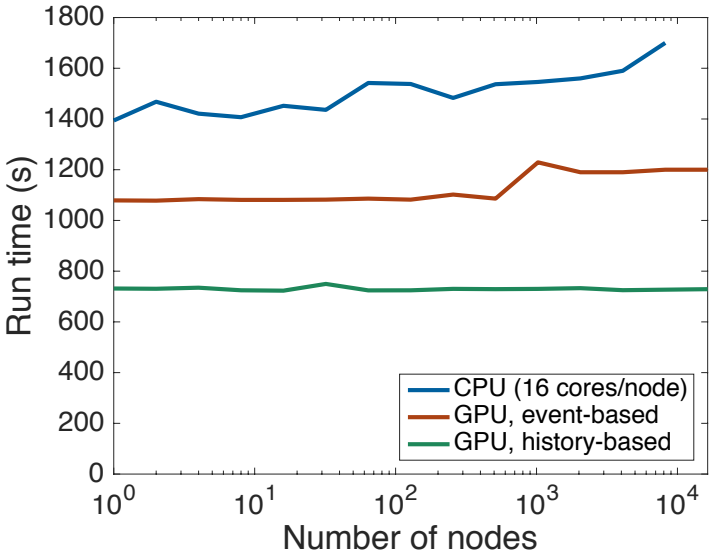


Figure 11: Run time as a function of node count for C5G7 problem on Titan.

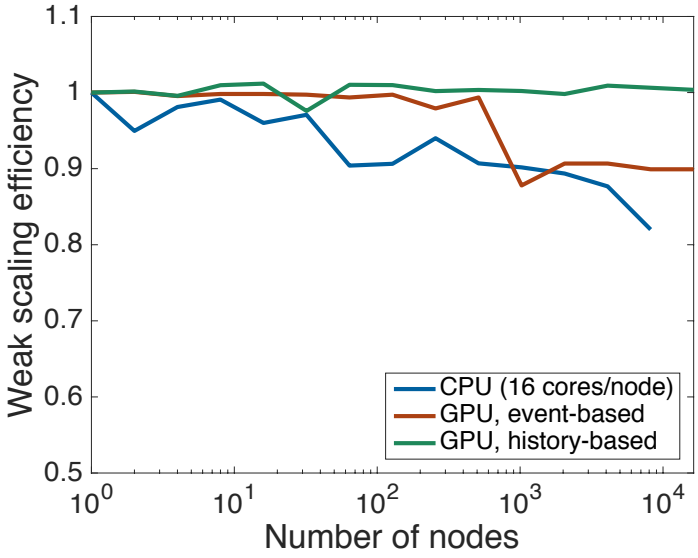


Figure 12: Weak scaling efficiency as a function of node count for C5G7 problem on Titan.

6. Conclusion

This paper describes the implementation of both history-based and event-based multigroup MC transport algorithms on the GPU within the Profugus code. A modified version of the history-based method is introduced which reduces the amount of high-level thread-divergence, improving efficiency on the GPU. Several algorithmic improvements were also introduced in the event-based method, including treatment of the source term as a new event type and a new approach to compute the indices of different event types without having to sort or otherwise rearrange particles between event cycles.

Performance studies on both fixed source and eigenvalue calculations were performed. Results indicate that the history-based approach outperforms the event-based method by factors of three to seven across the range of problems considered. The primary reason for this performance difference seems to be that while the event-based method does a better job of reducing thread divergence, it results in a less coherent memory access pattern which ultimately has a more signficant impact on performance. A large-scale parallel scaling study was performed on the Titan supercomputer using nearly the full machine. This study indicated that the GPU approaches show better parallel scaling effiency than the CPU version of Profugus due to the smaller number of MPI tasks required. It is expected that the parallel scaling of a CPU-based code using a hybrid MPI/threading approach would be similar to that of the GPU algorithms.

The primary focus for future investigations will be to extend the algorithms described here to the continous-energy case within the Shift MC transport code. Continuous-energy poses numerous additional challenges on the GPU due to the greatly increased volume of data required, as well as increased thread divergence due to the complex logic involved in computing cross sections during the transport process. Despite these issues, many of the conclusions from this work are expected to carry over to the continuous-energy case. Additionally, there are numerous applications in which continuous-energy data are not needed, but the use of MC is still preferred over deterministic methods due to improved treatment of the space and angle portions of phase space.

Acknowledgments

This research was sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U. S. Department of Energy. This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations—the Office of Science and the National Nuclear Security Administration—responsible for the planning and preparation of a capable exascale ecosystem—including software, applications, hardware, advanced system engineering, and early testbed platforms—to support the nation's exascale computing imperative. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] F. Brown, W. Martin, Monte Carlo methods for radiation transport analysis on vector computers, Progress in Nuclear Energy 14 (3) (1984) 269–299.

[2] R. Bergmann, J. Vujić, Algorithmic choices in WARP – a framework for continuous energy Monte Carlo neutron transport in general 3D geometries on GPUs, Annals of Nuclear Energy 77 (2015) 176–193.

[3] X. Xu, et al., ARCHER, a new Monte Carlo software tool for emerging heterogeneous computing environments, Annals of Nuclear Energy 82 (2015) 2–9.

[4] D. Ozog, A. Malony, A. Siegel, A performance analysis of SIMD algorithms for Monte Carlo simulations of nuclear reactor cores, in: IEEE 29th International Parallel and Distributed Processing Symposium, Hyderabad, India, 2015.

[5] T. Scudiero, Monte Carlo neutron transport: Simulating nuclear reactions one neutron at a time, in: GPU Technology Conference, San Jose, CA, 2014.

[6] T. Pandya, S. Johnson, T. Evans, G. Davidson, S. Hamilton, A. Godfrey, Implementation, capabilities, and benchmarking of Shift, a massively parallel Monte Carlo radiation transport code, Journal of Computational Physics 308 (2016) 239–272.

[7] SCALE: A Comprehensive Modeling and Simulation Suite for Nuclear Safety Analysis and Design, Tech. Rep. ORNL/TM-2005/39, Version 6.1, Oak Ridge National Laboratory (2011).

[8] J. Fleck, Jr., J. Cummings, An implicit Monte Carlo scheme for calculating time and frequency dependent nonlinear radiation transport, Journal of Computational Physics 8 (1971) 313–342.

[9] Oak Ridge Leadership Computing Facility, Titan Cray XK7, https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/ (August 2016).

[10] Profugus: A set of radiation transport mini-applications used for performance optimization on HPC systems, https://github.com/ORNL-CEES/Profugus.

[11] CUDA C programming guide, Tech. Rep. PG-02829-001_v7.5, NVIDIA (2015).

[12] The OpenACC application programming interface, Tech. Rep. version 2.5, OpenACC-standard.org (October 2015).

[13] H. C. Edwards, C. Trott, Kokkos, manycore device performance portability for C++ HPC applications, in: GPU Technology Conference, San Jose, CA, 2015.

[14] R. Hornung, J. Keasler, The RAJA portability layer: Overview and status, Tech. Rep. LLNL-TR-661403 (September 2014).

[15] P. Romano, B. Forget, Parallel fission bank algorithms in Monte Carlo criticality calculations, Nuclear Science and Engineering 170 (2012) 125–135.

[16] NVIDIA Tesla P100: The most advanced datacenter accelerator ever built, Tech. Rep. WP-08019-001_v01.1, NVIDIA.

[17] P. Romano, et al., Data decomposition of Monte Carlo particle transport simulations via tally servers, Journal of Computational Physics 252 (2013) 20–36.

[18] E. E. Lewis, et al., Benchmark on deterministic calculations without spatial homogenization: MOX fuel assembly 3-D extension case, Tech. Rep. NEA/NSC/DOC(2005)16, OECD/NEA (2005).

[19] E. Zhang, Y. Jiang, Z. Guo, K. Tian, X. Shen, On-the-fly elimination of dynamic irregularities for GPU computing, in: Sixteenth international conference on architectural support for programming languages and operating systems, Newport Beach, CA, 2011, pp. 369–380.

[20] B. Rearden, R. Lefebvre, J. Lefebvre, K. Clarno, M. Williams, L. Petrie, U. Mertyurek, Modernization enhancements in SCALE 6.2, in: PHYSOR 2014 - The Role of Reactor Physics Toward a Sustainable Future, Kyoto, Japan, 2014.

[21] R. Bleile, et al., Investigation of portable event-based Monte Carlo transport using the NVIDIA Thrust library, Transactions of the American Nuclear Society 114 (2016) 941–944.

[22] J. Cheng, M. Grossman, T. McKercher, Professional CUDA C Programming, John Wiley & Sons, Inc., Indianapolis, IN, 2014.

[23] T. Scudiero, GPU memory bootcamp II: Beyond best practices, in: GPU Technology Conference, San Jose, CA, 2015.

[24] Tuning CUDA applications for Kepler, Tech. Rep. DA-06288-001_v8.0, NVIDIA (2017).

[25] X. Du, T. Liu, W. Ji, X. Xu, F. Brown, Evaluation of vectorized Monte Carlo algorithms on GPU's for a neutron eigenvalue problem, in: International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering, Sun Valley, ID, 2013.

[26] V. Volkov, Better performance at lower occupancy, in: GPU Technology Conference, San Jose, CA, 2010.

[27] R. C. Bleile, P. S. Brantley, M. J. O'Brien, H. Childs, Algorithmic improvements for portable event-based Monte Carlo transport using the

NVIDIA Thrust library, Transactions of the American Nuclear Society 115 (2016) 535–538.

[28] Thrust quick start guide, Tech. Rep. DU-06716-001_v8.0, NVIDIA (June 2017).

[29] IBM Power System S822LC for high performance computing introduction and technical overview, Tech. Rep. REDP-5405-00, IBM (October 2016).

[30] K. Kobayashi, Proposal for 3D radiation transport benchmarks for simple geometries with void region, Progress in Nuclear Energy 39 (2001) 119–144.

[31] P. Romano, A. Siegel, Limits on the efficiency of event-based algorithms for Monte Carlo neutron transport, Nuclear Engineering and Technology 49 (6) (2017) 1165–1171.

[32] M. A. Smith, E. E. Lewis, B. C. Na, Benchmark on deterministic transport calculations without spatial homogenization – a 2-D/3-D MOX fuel assembly benchmark (C5G7 MOX benchmark), Tech. Rep. NEA/NSC/-DOC(2003)16, OECD/NEA (2003).

[33] C. Cathalau, J. C. Lefebvre, J. P. West, Proposal for a second stage of the benchmark on power distributions within assemblies, Tech. Rep. NEA/NSC/DOC(96)2, Rev. 2, OECD/NEA (1996).