SANDIA REPORT

SAND2018-10539
Unlimited Release
Printed September 2018

Coupled Electron-Photon Monte Carlo Radiation Transport for Next-Generation Computing Systems

Kerry L. Bossler

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology and Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Rd
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online ordering: https://classic.ntis.gov/help/order-methods/

SAND2018-10539
Unlimited Release
Printed September 2018

Coupled Electron-Photon Monte Carlo Radiation Transport for Next-Generation Computing Systems

Kerry L. Bossler
Radiation Effects Theory Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1179

Abstract

Traditional Monte Carlo particle transport codes are expected to run inefficiently on next-generation architectures as they are memory-intensive and highly divergent. Since electrons and photons also behave differently, the future for coupled electron-photon radiation transport looks even worse. This project describes preliminary efforts to improve the performance of Monte Carlo particle transport codes when using accelerators like the graphics processing unit (GPU). Two key issues are addressed: how to handle memory-intensive tallies, and how to reduce divergence. Tallying on the GPU can be done efficiently by post-processing particle data, or by using a feature called warp shuffle for summing scores in parallel during the simulation. Reducing divergence is possible by using an event-based algorithm for particle tracking instead of the traditional history-based one. Although performance tests presented in this work show that the history-based algorithm generally outperformed the event-based one for simple problems, this outcome will likely change as the complexity of the code increases.

Acknowledgments

The author would like to thank Ron Kensek (1341) and Greg Valdez (1341) for their guidance and feedback, as well as Frank Angers from the University of Michigan for obtaining the Savannah v0.2 performance results presented in this report.

Contents

List of Figures

List of Tables

This page intentionally left blank.

Nomenclature

Acronyms

CPU central processing unit

GPU graphics processing unit

ITS Integrated Tiger Series

KDE kernel density estimator

SIMT single instruction, multiple thread

SM streaming multiprocessor

Abbreviations

GB gigabytes

kB kilobytes (= 1024 bytes)

MB megabytes

GPU Abbreviations

K40 NVIDIA® Tesla® K40 GPU (Kepler™ architecture)

K80 NVIDIA® Tesla® K80 GPU (Kepler™ architecture)

K5200 NVIDIA Quadro® K5200 GPU (Kepler™ architecture)

P100 NVIDIA® Tesla® P100 GPU (Pascal™ architecture)

Symbols

C_i number of events processed for the i^{th} particle history

d_{ic} length of the c^{th} particle track processed from the i^{th} history

N number of particle histories

N_{out} number of particles that escape a geometric domain

V volume of a region of interest

w_{ic} particle weight for the c^{th} event processed from the i^{th} history

x spatial coordinate on the x-axis

$\hat{x}$ mean value for a physical quantity of interest

x_{ic} contribution to the mean value for the i^{th} history and c^{th} event

$\hat{\phi}$ estimate of the particle flux

μ linear attenuation coefficient for photon attenuation

Chapter 1

Introduction

Exascale supercomputers will likely consist of heterogeneous architectures, combining multi-core central processing units (CPUs) with accelerators such as graphics processing units (GPUs). As of June 2018, three of the top five supercomputers in the world use GPUs based on the latest NVIDIA Tesla architecture [1]. This includes both Summit and Sierra, the two newest systems hosted at national laboratories within the Department of Energy. Top supercomputers like Summit and Sierra rely on GPUs to boost performance because they offer high compute power at a lower energy cost than equivalent CPU-based systems.

For an application to run effectively on a GPU-based system, it needs to implement algorithms that can take advantage of the single instruction, multiple thread (SIMT) architecture. The best algorithms for a GPU are therefore highly data-parallel with few or no divergent paths in the code. Unfortunately, traditional Monte Carlo particle transport codes involve algorithms that are inherently memory-intensive and highly divergent. The divergence problem is exacerbated further in coupled electron-photon radiation transport, since both electrons and photons are tracked and these two particle types behave differently. At Sandia National Laboratories, one of the workhorse codes that performs coupled electron-photon radiation transport for directed nuclear stockpile work is the Integrated Tiger Series (ITS) [2]. As a legacy code based on traditional Monte Carlo algorithms, ITS is not expected to work well on next-generation architectures that include GPUs. To take full advantage of GPU-based systems, application codes like ITS will need to address both memory usage and divergence issues.

This project describes preliminary efforts to improve the performance of Monte Carlo particle transport codes on the GPU. Two key issues are addressed: how to handle memory-intensive data such as tallies, and how to reduce divergence in the core transport algorithm. In this introductory chapter, first an overview of the NVIDIA GPU architecture is provided in Section 1.1. Then, the general Monte Carlo particle transport algorithm is described in Section 1.2. Finally, Section 1.3 presents an outline of the scope of this work. A glossary containing the GPU computing and Monte Carlo particle transport terms used in this report can be found on page 57.

1.1 NVIDIA GPU Architecture

The general NVIDIA GPU architecture is built around a scalable array of multithreaded streaming multiprocessors (SMs), each designed to execute instructions for hundreds of threads concurrently [3]. Parallel work to be executed on the SIMT architecture of an NVIDIA GPU can be written using the CUDA® programming model also developed by NVIDIA. CUDA includes C/C++ language extensions that can be used to execute instructions on the GPU, or to transfer data between the CPU and the GPU. Executing instructions on the GPU is done by launching what is called a CUDA kernel. Each CUDA kernel that is launched breaks down the work into multiple thread blocks that are then distributed to all the available SMs. The SMs then process each thread block by creating, scheduling, and executing groups of 32 threads known collectively as a warp. All threads in a warp must execute single instructions concurrently, usually on different data sets read from memory. While individual threads are allowed to branch and execute instructions independently from the others, this branch divergence forces the code to become serialized and can have a significant impact on the performance of a CUDA kernel. Therefore, the SIMT architecture of an NVIDIA GPU is operating at its optimal efficiency when there is no branch divergence within a warp.

1.1.1 Device Memory Hierarchy

Data used by a thread to execute an instruction on an SM must first be read from one of the many memory spaces available on an NVIDIA GPU. This memory hierarchy ranges from register memory assigned to individual threads, up to global memory that is accessible to the CPU host and all the threads being processed by a CUDA kernel. A general summary of the different memory spaces is shown in Table 1.1. Each memory space listed in Table 1.1 comes with unique advantages and disadvantages, so some thought must go into choosing the right type to use for different data accesses.

Register Memory: Since register memory is located on-chip, it provides the fastest access of all the options that are available. However, register memory is a limited resource that must be shared by all the threads assigned to an SM.

Local Memory: Local memory is much slower than register memory because it is located off-chip, so it is important to minimize its usage to obtain optimal performance. There are two situations to avoid that will cause the compiler to use local memory. The first occurs when a CUDA kernel requires more registers than are available to an SM, which causes the excess data to spill over into local memory. The second occurs when an array has been declared inside a CUDA kernel and needs to be accessed using dynamic indexing. In other words, the access pattern to the array cannot be determined at compile time.

CUDA® is a registered trademark of NVIDIA Corporation.

Shared Memory: Like register memory, shared memory is also located on-chip. The primary advantage of this memory space is that it can be used by all the threads in a single block, which enables efficient communication between the threads. Disadvantages of shared memory are that it is limited to 48 kB per thread block, and also that synchronization across all the threads in the block may be required.

Global Memory: Global memory is the largest of all the options, but is also the slowest because it is located off-chip. Most of the data that CUDA kernels need to access throughout the entire duration of the host program will be stored in global memory. A best practice for achieving optimal performance is to minimize the number of times global memory needs to be read or written by a CUDA kernel [4].

Constant Memory: Like global memory, constant memory is located off-chip. However, constant memory differs from global memory in that it is limited to 64 kB and is cached on-chip for efficient read-only access. The optimal use case for constant memory is when all threads in a warp read from the same location, which makes it as fast as register memory [4].

Texture Memory: Texture memory is another form of read-only memory located off-chip that is cached on-chip. As it was designed for graphics applications, texture memory works best when the access pattern involves spatial locality (i.e., all threads in a warp read from locations that are close to one another).

Table 1.1: Device memory hierarchy of an NVIDIA GPU [4].

Memory Space	Location	Access Type	Scope	Lifetime
Register	On-chip	Read/Write	One Thread	Thread
Local	Off-chip	Read/Write	One Thread	Thread
Shared	On-chip	Read/Write	One Thread Block	Block
Global	Off-chip	Read/Write	All Threads & Host	Host Allocation
Constant	Off-chip	Read-Only	All Threads & Host	Host Allocation
Texture	Off-chip	Read-Only	All Threads & Host	Host Allocation

1.1.2 Atomic Functions and Warp Shuffle

Writing data to global or shared memory on the GPU is not as straightforward as writing data to memory in a serial application that runs on the CPU. Since multiple threads can access the same memory address at the same time, this causes a race condition that will produce inconsistent results because the order in which the data is read and updated is not guaranteed. To avoid race conditions, threads on an NVIDIA GPU can use atomic functions that perform a read-modify-write operation in either global or shared memory. Atomic

functions are executed as a serial operation, which means that if one thread is executing an atomic function, then the other threads in the warp must wait until it is finished before they can continue executing their instructions. One of the most commonly used atomic functions is atomicAdd, which will first read a value from memory, then compute the sum of that value with a new value, and finally write the sum of the old value and the new value into memory at the same address. An alternative to using atomicAdd for adding up numbers stored on individual threads in a warp is to use an NVIDIA GPU feature called warp shuffle, which allows up to 32 threads to simultaneously exchange or broadcast data. Warp shuffle can be used to implement an efficient parallel reduction across all the threads in a warp [5], which is shown graphically in Figure 1.1.[1]

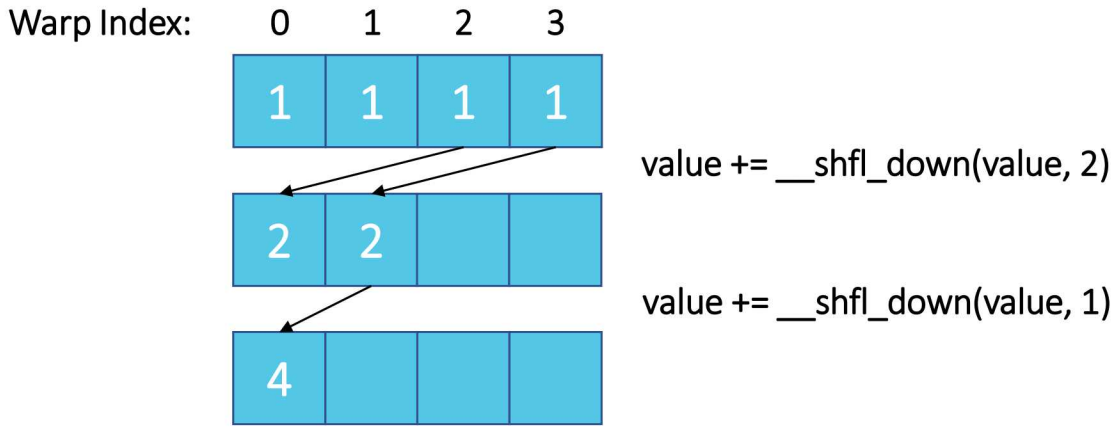

Figure 1.1: Graphical representation of a parallel reduction using NVIDIA's warp shuffle feature to add up individual values stored on four threads.

1.1.3 NVIDIA GPU Options

Even though all NVIDIA GPUs use an SIMT architecture and the same device memory hierarchy, there can be some significant differences between the various cards that are available. The different performance studies presented in this report all use at least one of the following NVIDIA GPUs: Quadro K5200, Tesla K40, Tesla K80, and Tesla P100. The three Tesla GPUs are dedicated accelerators used for scientific computing applications, whereas the one Quadro GPU was designed to provide graphics for a desktop workstation. Table 1.2 summarizes the key differences of these four GPUs.

[1]Figure 1.1 is based on the warp shuffle syntax in CUDA 8, which requires that all the threads in a warp participate in the shuffle. In CUDA 9, the syntax and behavior of warp shuffle has changed so that the programmer can select what threads in the warp will participate.

Table 1.2: Comparison of different NVIDIA GPU architectures.

Specification	**K5200**	**K40**	**K80**	**P100**
# GPUs per Card	1	1	2	1
CUDA Compute Capability	3.5	3.5	3.7	6.0
CUDA Cores	2304	2880	2496	3584
Streaming Multiprocessors	12	15	13	56
GPU Clock Rate (MHz)	771	745	824	1481
Single Precision TeraFLOPS	–	5.0	8.7[1]	10.6
Double Precision TeraFLOPS	–	1.7	2.9[1]	5.3
Global Memory (MB)	8125	11,441	11,441	16,281
Memory Bandwidth (GB/s)	192	288	240	732
Device to Host Bandwidth (GB/s)	∼3	∼9	∼9	∼11

[1]Single & double precision performance for K80 is combined value for both GPUs on the card.

1.2 Monte Carlo Particle Transport

Monte Carlo particle transport is a stochastic method used to model the behavior of particles in one or more fixed background materials. A finite number of particles are first generated from a source, then experience a randomly-determined sequence of events that changes their energy and/or direction until they either get absorbed or leave the system. This sequence of events that each particle experiences is also known as a history.

1.2.1 History-Based and Event-Based Transport Algorithms

Two vastly different approaches can be used to implement the core transport algorithm of a Monte Carlo particle transport code. The history-based transport algorithm shown in Figure 1.2 is the traditional approach, which follows the path of individual particles. Each history representing one particle is processed until it gets terminated, then the next history can begin. The simulation ends when all particles have been terminated. In production codes, the history-based transport algorithm is implemented using the Message Passing Interface (MPI) standard to distribute groups of particles to different compute nodes for processing. On today's high performance computers, this approach is embarrassingly parallel. However, for next-generation architectures that rely on SIMT to increase computational power, the traditional history-based approach can be less effective due to the possibility of high divergence in the paths of individual particles.

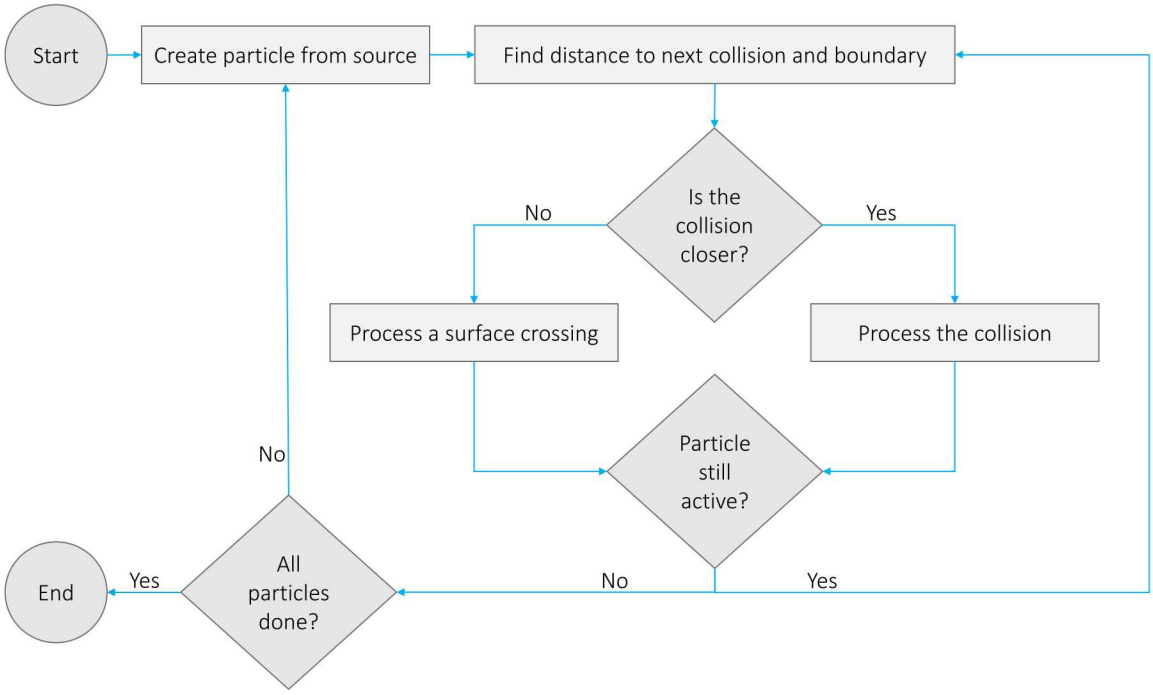

Figure 1.2: General workflow for a history-based transport algorithm.

An effort to reduce the branch divergence of the history-based transport algorithm was first proposed for vector processors by Brown and Martin back in 1984 [6]. Designated the event-based transport algorithm, this alternative approach sorts particles into groups according to the type of event they are expected to experience next. The general workflow for the event-based transport algorithm is shown in Figure 1.3. Instead of following individual particles, the event-based approach splits up the transport algorithm into a series of distinct events, then processes all particles for one event before moving onto the next one. Defining events to use varies with the type of Monte Carlo particle transport being implemented, but could include things such as creation, absorption, scattering, surface crossing, etc.

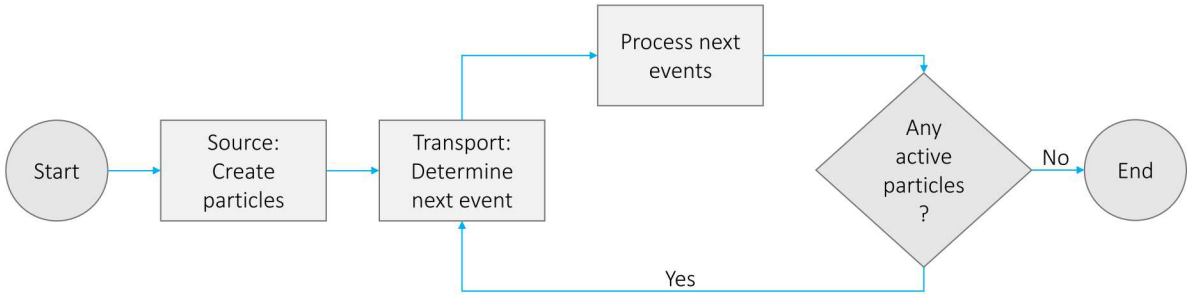

Figure 1.3: General workflow for an event-based transport algorithm.

1.2.2 Tallies

During the Monte Carlo particle transport simulation, each history is assigned a score using a statistical estimator to convert the average particle behavior into a physical quantity of interest. Scores for each history are accumulated in what is called a tally:

$$\hat{x} = \frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C_i} x_{ic}, \tag{1.1}$$

where $\hat{x}$ is the mean value for some physical quantity of interest, N is the number of histories, C_i is the number of events for the i^{th} history, and x_{ic} is the contribution for the i^{th} history and c^{th} event. Each region of interest defines its own tally using Equation 1.1, which can also be split up into additional bins to separate contributions for different energies, angles, or time. Only contributions that occur within the region of interest will be added to its tally. If the regions of interest are defined as elements on a mesh, then the tallies for all the elements are collectively known as a mesh tally.

One of the most commonly used tallies in Monte Carlo particle transport is the particle flux tally, which measures the total length traveled by the particles in a region per unit volume and time ($\text{cm}^{-2}\,\text{s}^{-1}$). Particle flux ϕ can be estimated using either a collision tally, which counts the number of collisions in a region, or a track length tally:

$$\hat{\phi} = \frac{1}{NV} \sum_{i=1}^{N} \sum_{c=1}^{C_i} w_{ic} d_{ic}, \tag{1.2}$$

where V is the volume of the region of interest, w_{ic} is the particle weight, and d_{ic} is the track length for the c^{th} particle track that is processed from the i^{th} history. The track length tally generally provides a more accurate result for the particle flux than the collision tally, especially in regions with few collisions. Note that to get the correct units for particle flux, Equation 1.2 must also be multiplied by the source strength (i.e., particles per second).

1.3 Scope of Work

Improving the performance of Monte Carlo particle transport codes on next-generation architectures is a challenging task with many unknowns. The goal of this project was therefore limited to evaluating two key issues for coupled electron-photon radiation transport on the GPU, namely how to handle memory-intensive tallies, and how to reduce divergence. Reducing divergence in particular may require a paradigm shift from traditional approaches used by most production Monte Carlo particle transport codes. Although the original intent was to focus on coupled electron-photon radiation transport, the following discussion is relevant for all types of Monte Carlo particle transport.

The structure of this report consists of two main chapters, as well as a concluding chapter with recommendations for future research needed to improve the performance of coupled electron-photon radiation transport codes on GPUs. Chapter 2 summarizes efforts to find effective methods for computing Monte Carlo tallies on the GPU, including multiple options for simple event counters and mesh tallies. Chapter 3 introduces Savannah, an exploratory event-based Monte Carlo particle transport mini-app that was developed to explore the event-based transport algorithm as an alternative to the history-based transport algorithm. Performance results using both transport algorithms in Savannah are compared for a number of different test cases and GPU architectures.

Chapter 2

Monte Carlo Tallies on the GPU

Tallies are one of the most frequently performed tasks across all the different variants of Monte Carlo particle transport codes. Although the structure of a Monte Carlo tally can vary significantly, depending on the physical quantity of interest, they generally involve computing and adding scores to one or more bins discretized by spatial location, energy, angle, and time. As multiple particles can contribute scores to the same tally bin, race conditions must be avoided to obtain valid results from a parallel calculation. In previous work, race conditions were avoided on the GPU by implementing tallies using either atomic functions [7–11] or tally replication [11–14]. Tally replication is also one of the most commonly used methods for implementing tallies on CPU-based high performance computing systems. However, tally replication is less attractive for GPU-based systems because memory is much more limited. Relying on atomic functions to reduce memory usage is not always the best option either, since any tally updates that result in a race condition will be serialized, and adding more serialization to a parallel calculation can significantly impact its overall performance.

In practice, the best approach to use for implementing a specific type of tally on the GPU depends on multiple factors such as its size, required precision, and its update frequency. This chapter summarizes efforts to find effective methods for computing different Monte Carlo tally types on the GPU, with Section 2.1 comparing five options for simple event counters, Section 2.2 describing an efficient implementation of the kernel density estimator (KDE) integral-track mesh tally [15], and Section 2.3 describing a different way of thinking about the conventional mesh tally.

2.1 Event Counters

Event counters are tallies used to count the number of occurrences of a specific event type, such as the number of particles that escape the problem domain, or the number of scattering events that occur within that domain. Each event counter only needs to store a single integer value in memory, which means that there are a wide variety of options available for implementing them on the GPU. Five different methods were compared for a simple photon escape tally and are described in detail in a separate publication [16]. Recommendations for the ideal use case for each method are summarized in Table 2.1.

Table 2.1: Different options for implementing event counters on the GPU.

Method	**Atomics**[1]	**Ideal Use Case**
Tally Replication	N/A	Small tally with high update frequency
Global Atomics	128 Global	Large tally with low update frequency
Shared Atomics	128 Shared 1 Global	Small tally with low update frequency
Warp Shuffle[2]	4 Global	Large integer tally with high update frequency
Block Reduction[2]	1 Global	Large floating-point tally with high update frequency

[1]Number of atomic function calls assuming 128 threads per block.

[2]Method uses NVIDIA's warp shuffle feature.

The best option to use for implementing event counters depends on the data type, update frequency, and memory availability. Tally replication will usually be the most performant option for all data types, especially when the event counters need to be updated frequently. However, as more memory is needed for other memory-intensive data in the Monte Carlo particle transport simulation (i.e., geometry and cross sections), then there are several alternatives that can also perform well on the GPU in certain situations. Using atomic functions on event counters stored in global memory can be effective when there is a low update frequency, which means that there will be few race conditions occurring to serialize the code. If the event counters are small enough, however, then performance can be improved by storing the tally data in shared memory. Atomic functions are more efficient when reading and writing to shared memory instead of global memory.

When event counters are too big for shared memory and need to be updated frequently, then the best option to use depends on the data type. For integers, the most effective alternative is to use the warp shuffle method. However, this approach still requires one atomic function call in global memory to update the tally with the contribution from each warp. For floating-point values it is better to use the block reduction method [16], as it reduces the number of atomic function calls to one per block instead of one per warp. The block reduction method does require synchronization across all the threads in the block, but this cost is lower than the cost of the additional floating-point atomic function calls.

2.2 KDE Integral-Track Mesh Tally

Event counters are simple tallies that do not require much memory unless tally replication is used. In contrast, mesh tallies can require a lot of memory to store not only the elements of the mesh, but also the tally structure for each element. As a result, tally replication is not likely feasible for implementing mesh tallies on the GPU. Furthermore, if GPU threads are

assigned to particles, then race conditions are likely as multiple particles could contribute to the same element in the mesh. An alternative approach for the KDE integral-track mesh tally is described in detail in a separate publication [17]. This alternative approach assigns GPU threads to nodes in the mesh, which means that no race conditions can occur because each mesh node only ever needs to access its own tally. In this report, a brief summary of the implementation and the significance of the results is presented.

GPU and serial CPU versions of the KDE integral-track mesh tally were implemented in a new tool called Rapid that can be linked to any Monte Carlo particle transport code. Rapid is currently linked to ITS and is able to process data transferred from ITS in two ways. The first way is to compute mesh tallies during the Monte Carlo transport simulation as event data is received from ITS. The second way is to store event data from ITS in a Hierarchical Data Format 5 (HDF5) file, then run Rapid by itself after the Monte Carlo particle transport simulation is complete to post-process mesh tallies directly from that file. Due to the large cost involved in memory transfers between the GPU and CPU, Rapid splits up the work for processing all mesh tallies into setup, compute, and finalize stages. The setup stage prepares the mesh tally, the compute stage updates the tally, and the finalize stage reports final tally results. This separation of tasks is performed whether Rapid is used for in-situ tallying or for post-processing tallying.

Results for one of the tests that were performed are shown in Table 2.2, which computed scores for a mesh with over 10 million nodes. Although the setup and finalize times are higher for the GPU implementation, this is expected because the GPU implementation of the KDE integral-track mesh tally needs to perform more work to transfer data between the CPU and the GPU. During the setup stage, nodal coordinates are copied into global memory and invariant parameters needed for the KDE integral-track estimator to compute the scores are copied into constant memory. During the finalize stage, tally results must be copied back to the CPU and resources allocated on the GPU must also be freed.

Table 2.2: Timing data for computing over 10 million scores for a KDE integral-track mesh tally on a Quadro K5200.

STAGE	**CPU (ms)**	**GPU (ms)**
Setup	7.81×10^4	7.87×10^4
Compute	1.46×10^4	5.80×10^1
Finalize	0.51×10^4	1.07×10^4
Total	$\mathbf{9.78 \times 10^4}$	$\mathbf{8.94 \times 10^4}$

Although computing scores for a KDE integral-track mesh tally does come with an increased cost with respect to setup and finalize stages, the GPU implementation was able to process over 10 million scores in the compute stage 250 times faster than the CPU implementation.

This compute stage is also the only one that is repeated, typically millions or billions of times in a single simulation. Therefore, the GPU implementation should rapidly start to outperform the serial CPU implementation as more scores are computed for large meshes. For example, taking the setup and finalize costs into consideration, tallying over 10 million scores 1000 times is expected to be about 100 times faster on the GPU than a serial CPU.

2.3 Stratified Sampling Mesh Tally

Being able to efficiently compute KDE mesh tallies on a GPU is useful, but this type of tally is not widely used in production Monte Carlo particle transport codes for real analysis work. The more conventional approach is to first define each region of interest as a histogram bin, and then apportion particle event data into those bins either deterministically or stochastically. For example, consider a track length mesh tally that uses Equation 1.2 to define a tally bin for every element in a mesh. The deterministic approach requires that each mesh element uses the exact length of the track that falls within it as its d_{ic} value. Alternatively, a stochastic approach called stratified sampling can be used. Stratified sampling divides the particle track into a fixed number of subtracks, or strata, with equal lengths. One random point is then chosen within each stratum, and the mesh element in which that point is located adds the fixed subtrack length to its tally bin instead of the exact track length.

Unfortunately, the stratified sampling mesh tally is not naturally data-parallel because the time it takes to locate the mesh element in which each random point is located can differ substantially. If each GPU thread was responsible for computing a tally score for a different particle track, this variation in the point-in-element search could introduce a significant amount of branch divergence. The following sections assess an alternative implementation of the stratified sampling mesh tally that assigns GPU threads to mesh elements instead of particle tracks. This implementation improves the data-parallelism of the conventional mesh tally by making it behave more like the KDE integral-track mesh tally from Section 2.2.

2.3.1 GPU versus CPU Implementation

GPU and serial CPU implementations of a stratified sampling mesh tally were added as new tally options to Rapid. Using Rapid provided the same common abstract framework that was used for the KDE integral-track mesh tally implementations. Like the KDE integral-track mesh tally, the GPU implementation of the stratified sampling mesh tally needs to perform additional work in the setup and finalize stages. During the setup stage, all the mesh data (i.e., minimum and maximum points of each element) is copied into global memory, and the number of strata to use is copied into constant memory. During the finalize stage, tally results must be copied back to the CPU and resources allocated on the GPU must also be freed. Although there are some differences in the setup and finalize stages that will impact overall performance, these two stages are only called once per batch of particles processed.

In contrast, the compute stage is called multiple times to compute tally scores for each event until all the particles have been removed from the system.

For the KDE integral-track mesh tally, both the GPU and CPU implementations used the same algorithm to compute the tally score. For the stratified sampling mesh tally, however, there is one major difference in how the GPU and CPU algorithms are implemented. Both implementations iterate over the number of strata points, determine a random point for each strata, and add a score to the tally for the mesh elements in which those points are located. The key difference is how the point-in-element search is performed. The CPU implementation uses a simple linear search that searches mesh elements in order until the one in which the random point exists is found. The GPU implementation takes a slightly different approach, since each GPU thread is assigned to a mesh element instead of a particle track. Each mesh element performs a check in parallel to see if the random point that was selected exists within their domain. Even though only at most one element will return true, this check is executed efficiently on the GPU because it is a highly data-parallel algorithm compared to the linear search approach.[1] The mesh element that contains the random point, if any, updates its tally. There are no race conditions when updating the tally, since each mesh element only needs to update its own tally bin in global memory.

2.3.2 Performance Tests

The performance of the compute stage of the stratified sampling mesh tally was tested using 15 different mesh configurations, which included 8 options for a 1D slab, 4 options for a 2D square, and 3 options for a 3D cuboid geometry. Elements for all mesh configurations were uniformly distributed over an extent ranging from -100,000 cm to 100,000 cm in each dimension. Table 2.3 summarizes the number of mesh elements defined for each of these mesh configurations. All 15 mesh configurations listed in Table 2.3 were tested against different numbers of strata points and the following five unique particle track cases:

1. All strata points located in the first mesh element.
2. All strata points located in the last mesh element.
3. Strata points distributed over multiple mesh elements in the forward direction.
4. Strata points distributed over multiple mesh elements in the backward direction.
5. All strata points located outside the mesh geometry.

Note that first and last mesh element is defined as the first and last element accessed in the data structure that stores the mesh in memory. This may not always be equivalent to the spatial location of the first and last mesh element in the physical mesh, especially for 2D and 3D mesh configurations.

[1] GPU implementation assumes that no points will fall on the boundaries of a mesh element.

Table 2.3: Mesh configurations used to test a stratified sampling mesh tally.

Configuration	Number of Mesh Elements			
	X	*Y*	*Z*	*Total*
1	1	-	-	1
2	10	-	-	10
3	100	-	-	100
4	1000	-	-	1000
5	10,000	-	-	10,000
6	100,000	-	-	100,000
7	1,000,000	-	-	1,000,000
8	10,000,000	-	-	10,000,000
9	10	10	-	100
10	100	100	-	10,000
11	1000	1000	-	1,000,000
12	3200	3200	-	10,240,000
13	10	10	10	1000
14	100	100	100	1,000,000
15	215	215	215	9,938,375

2.3.3 Results

All performance tests described in Section 2.3.2 were run on a desktop Linux workstation with Intel® Xeon® E5-2697 v3 (2.60 GHz) CPUs and one NVIDIA Quadro K5200 GPU. Each test was repeated 10 times on the GPU and CPU to compute average runtimes. Figure 2.1 summarizes the results from all the performance tests as a histogram of the speedup of the GPU implementation over the serial CPU implementation. Most of the 375 variations used to create the histogram in Figure 2.1 were computed more efficiently on the GPU than the CPU. Only 54 (∼14%) of these variations were faster on the CPU, which occurred when there were not enough strata points or mesh elements to offset the overhead of computing tally scores on the GPU, or when the entire particle track fell inside the first mesh element.

Intel® and Xeon® are registered trademarks of Intel Corporation.

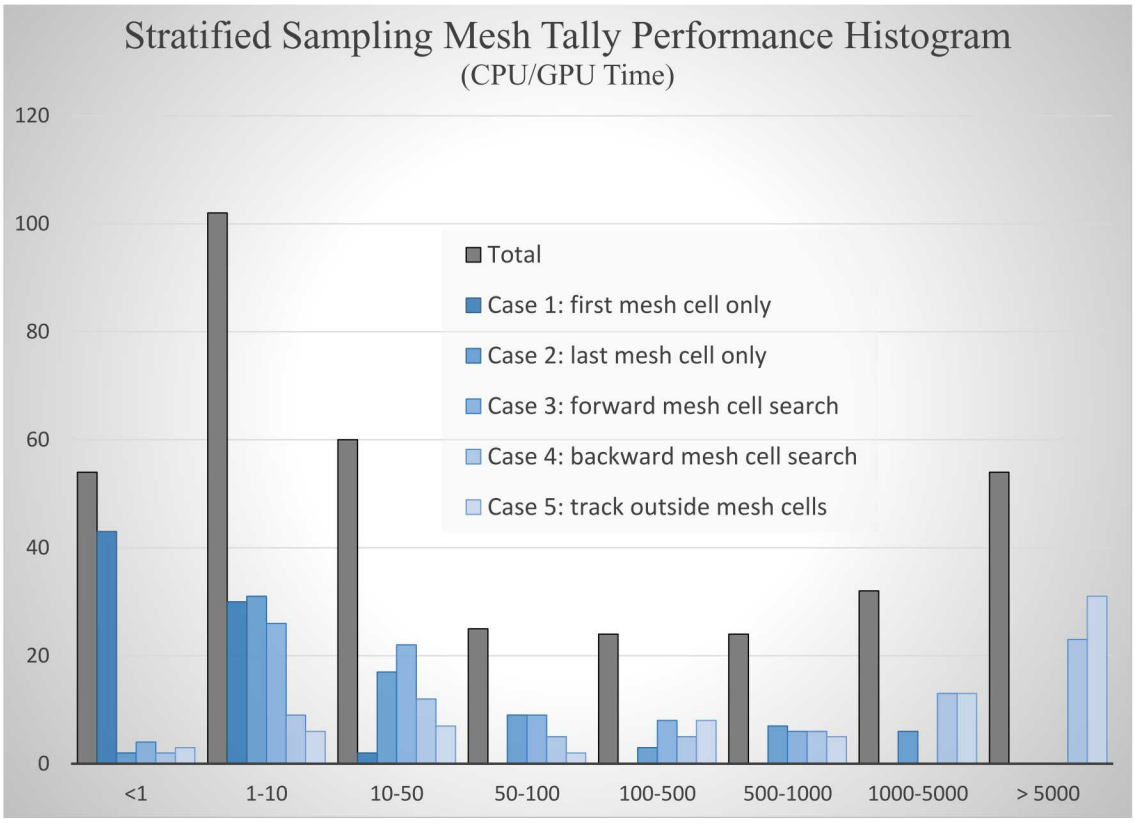

Figure 2.1: Speedup of GPU over CPU when computing tally scores using a stratified sampling mesh tally. Histogram compiled from 375 variations, including 15 uniformly distributed mesh configurations, 5 different numbers of strata, and 5 unique particle track cases.

Figure 2.2 plots GPU speedup as a function of the number of 1D mesh elements when all strata points fall inside the first mesh element. The performance of the GPU implementation relative to the CPU implementation gets worse as more elements are added to the mesh, no matter how many strata points are used. This behavior is not surprising given the different implementations of the point-in-element searches. The CPU implementation is not affected by the increase in mesh elements because it only needs to check the first mesh element for every strata point. However, the GPU implementation checks every mesh element in parallel for every strata point, which gets more expensive as more mesh elements are added. One possibility for improving the GPU implementation for this test case is to check to see if the entire particle track falls inside one mesh element before processing the strata points. If all strata points exist inside one mesh element, then no further processing would be needed.

Although the GPU implementation performed poorly when all strata points fell inside the first mesh element, the situation is reversed when substantial searches through large meshes are required. There were 54 variations where the GPU implementation was more than 5000 times faster than the CPU implementation, which included 31 cases where all strata points

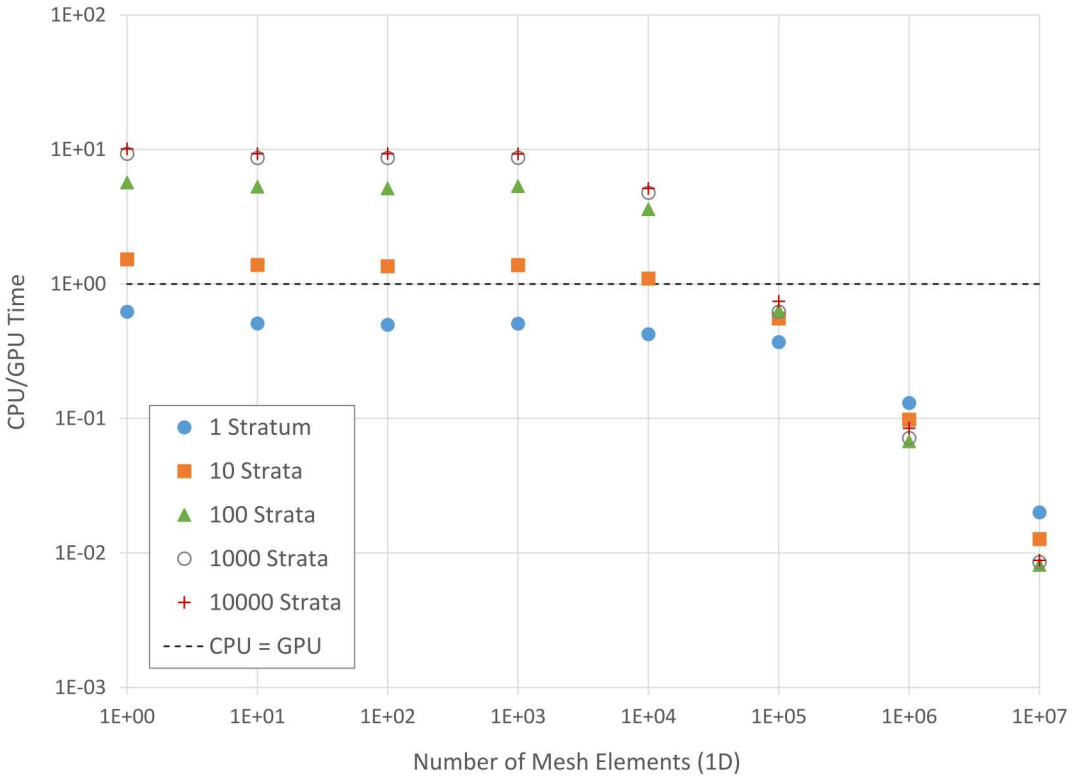

Figure 2.2: GPU performance of a stratified sampling mesh tally when all strata points are inside the first mesh element.

were outside the mesh. Figure 2.3 plots GPU speedup as a function of the number of 1D mesh elements when all strata points are outside the mesh.

In contrast to Figure 2.2, Figure 2.3 shows that the performance of the GPU implementation relative to the CPU implementation improves as more elements are added to the mesh. This behavior can also be explained by the different implementations of the point-in-element searches. The CPU implementation searches through every mesh element to determine that each strata point falls outside the mesh, which gets much more expensive as more more mesh elements are added. The GPU implementation also gets more expensive as more mesh elements are added, but the runtime is equivalent to the case where all strata points are inside the first mesh element. In other words, the GPU implementation is invariant to the value and location of the particle track, whereas the CPU implementation is highly sensitive to how much of the mesh needs to be searched.

The invariance of the GPU implementation to the particle track can be seen more clearly in Table 2.4 on page 30, which shows the timing results for a 1D mesh with 10^7 elements. When the number of mesh elements and strata points are fixed, the runtime of the GPU implementation is always approximately the same. When the number of strata points increases, the runtime also increases linearly. This behavior is caused by the GPU threads

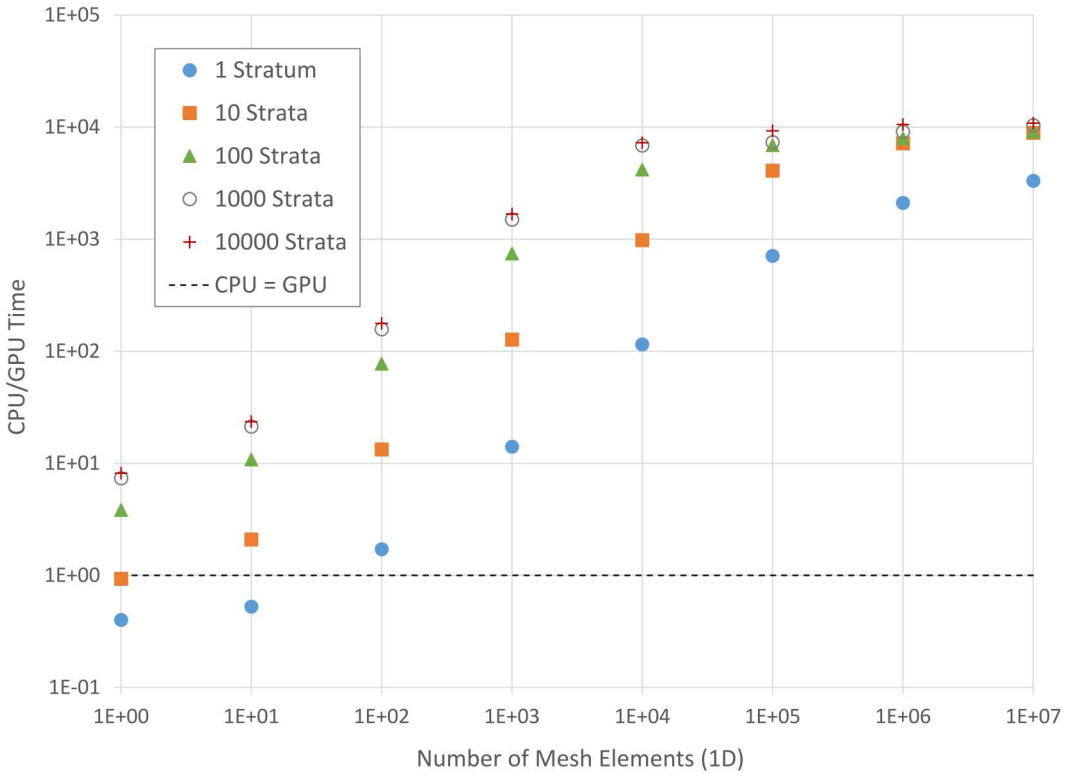

Figure 2.3: GPU performance of a stratified sampling mesh tally when all strata points are outside the mesh.

being mapped to mesh elements. Performance could therefore be improved by also parallelizing the strata points, which could be done by launching secondary CUDA kernels from the primary CUDA kernel.

For the CPU implementation, Table 2.4 verifies that the pattern in which the mesh elements are checked has a big impact on its performance. The most efficient case, as was previously discussed, is clearly the one where all strata points fall inside the first mesh element. For Cases 4 and 5, which both require multiple searches of most or all of the mesh, the performance of the CPU implementation is consistently about 10^4 times worse than the GPU implementation. Note, however, that the runtimes for the CPU could be improved by using a more efficient searching algorithm than the linear search.

Whereas the runtime of the GPU implementation always increased linearly with the number of strata points, this was only true for Cases 1, 4, and 5 with the CPU implementation. Cases 2 and 3 performed mostly the same for all of the strata points considered, with the GPU implementation approaching the same runtime at 10^4 strata points. The reason for this behavior is that the linear search algorithm on the CPU stores the last mesh element index that was found. For Case 2, which involves all strata points inside the last mesh element, the entire mesh therefore only needs to be searched once. Similarly for Case 3, which involves a

Table 2.4: GPU and CPU timing results using a stratified sampling mesh tally on a 1D mesh with 10^7 elements.

GPU Timing Results (ms)					
Strata	*Case 1*	*Case 2*	*Case 3*	*Case 4*	*Case 5*
10^0	3.074×10^0	3.195×10^0	3.147×10^0	3.082×10^0	3.142×10^0
10^1	1.170×10^1	1.153×10^1	1.152×10^1	1.171×10^1	1.171×10^1
10^2	1.131×10^2	1.161×10^2	1.110×10^2	1.164×10^2	1.146×10^2
10^3	1.010×10^3	9.896×10^2	1.011×10^3	9.943×10^2	1.013×10^3
10^4	9.692×10^3	9.692×10^3	9.697×10^3	9.694×10^3	9.697×10^3
CPU Timing Results (ms)					
Strata	*Case 1*	*Case 2*	*Case 3*	*Case 4*	*Case 5*
10^0	6.168×10^{-2}	1.042×10^4	1.408×10^3	9.036×10^3	1.043×10^4
10^1	1.488×10^{-1}	1.042×10^4	1.039×10^4	9.373×10^4	1.041×10^5
10^2	9.223×10^{-1}	1.042×10^4	1.036×10^4	1.042×10^6	1.049×10^6
10^3	8.637×10^0	1.055×10^4	1.048×10^4	1.047×10^7	1.050×10^7
10^4	8.554×10^1	1.055×10^4	1.058×10^4	1.045×10^8	1.050×10^8

forward mesh search to locate each strata point, the next search starts at the mesh element found by the last search. Without this optimization, the performance results for Cases 2 and 3 would likely be similar to the performance of Cases 4 and 5.

Chapter 3

Event-Based Monte Carlo Particle Transport

The transport algorithm used by a Monte Carlo particle transport code defines how the simulation will track particles from a source to produce results of interest to the user. On the CPU, using a history-based approach has been the method of choice for decades. With the introduction of scientific computing on the GPU, however, there has been renewed interest in the event-based transport algorithm first introduced in the 1980s for vector processors [6]. Adopting an event-based transport algorithm on next-generation architectures that include GPUs would represent a paradigm shift for production Monte Carlo particle transport codes.

Prior research efforts within the medical physics community have experimented with using the GPU for coupled electron-photon radiation transport [18–20]. Although all these implementations used a history-based approach, some effort at reducing divergence was done by the GPU Monte Carlo Dose (GPUMCD) code team by processing electrons and photons separately [19]. Only one work is known to have attempted using the event-based approach for coupled electron-photon radiation transport: the Vectorized Dose Planning Method (V-DPM) code from 2003 [21]. However, V-DPM just vectorized the electron transport, not the photon transport. In addition, due to hardware limitations of the vector processors that were used at the time, only groups of four electrons could be processed concurrently within the vectorized portions of the code.

Since today's GPUs can process thousands of threads simultaneously, more recent work has reconsidered the event-based transport algorithm for nuclear engineering applications. In particular, the WARP code was able to successfully implement event-based transport for solving neutron eigenvalue problems on the GPU [7]. However, WARP only compared the performance of their event-based implementation to history-based codes developed for the CPU, such as MCNP® [22] and Serpent [23]. Other efforts that have compared history-based and event-based neutron transport on the GPU in the same code have reported varying results. One of these efforts is the Monte Carlo neutron transport code called Guardyan, which is developed at the Budapest University of Technology and Economics. The Guardyan team reported speedups of 1.5 to 2 times for their event-based implementation, but also

MCNP® is a registered trademark of Los Alamos National Security, LLC, manager and operator of Los Alamos National Laboratory.

noted that a light water reactor assembly test case that they ran resulted in a 1.5 times slowdown [24]. Slowdowns of 3 to 7 were also reported in a research Monte Carlo neutron transport code developed at Oak Ridge National Laboratory [11]. The most recent work, also performed at Oak Ridge, reported a 2 times speedup for the event-based implementation in Shift, which is a production Monte Carlo neutron transport code [25].

This chapter summarizes efforts in comparing the history-based and event-based transport algorithms implemented in a new research code called Savannah. Since recent work on GPUs for Monte Carlo particle transport has focused on neutrons, the ultimate goal of Savannah is to assess the effectiveness of using an event-based transport algorithm for coupled electron-photon radiation transport on the GPU. Section 3.1 introduces the current version of Savannah, and Section 3.2 compares the performance of the history-based and event-based implementations for two types of problems: photon attenuation and isotropic scattering.

3.1 Savannah: Exploratory Event-Based Monte Carlo Particle Transport

Savannah is a mini-app designed to explore the effectiveness of the event-based transport algorithm compared to the traditional history-based one. Although the ultimate goal of Savannah is to study coupled electron-photon radiation transport, the current version only implements simple photon transport. Photons that are created from a point source are transported through a 1D slab geometry. These photons can either get absorbed, undergo isotropic scattering, or escape the problem domain. The following sections describe key implementation details of Savannah v0.2, which includes the execution model, how particles are stored, the random number generator, the types of events and tallies that are available, and a method called particle remapping expected to improve the performance of the event-based transport algorithm on the GPU.

3.1.1 Execution Model

After processing the input file and extracting the input options, Savannah creates an execution space that defines where the particle transport simulation will be run. This abstract concept of an execution space is also used by the Kokkos programming model [26]. Although Savannah includes separate execution spaces for serial CPU and the GPU, this report focuses on the GPU implementation. When initializing the GPU execution space, data describing the 1D slab geometry, cross sections, and source definition are all copied into constant memory. Constant memory is used for this data as it is read-only and, in general, all the threads in a warp need to access the data at the same time. Once the GPU execution space has been initialized, the next stage is to run the core transport simulation. The transport stage is made up of one or more CUDA kernels depending on whether the algorithm used is history-based (see Figure 1.2) or event-based (see Figure 1.3). For

history-based transport, there is only one CUDA kernel called the Big Kernel that processes one photon per GPU thread from its birth until its death. For event-based transport, the implementation of the Big Kernel is split up into four main CUDA kernels:

- **Source Kernel**: Creates photons from a common point source definition.
- **Transport Kernel**: Identifies next event for photons and moves them to that event.
- **Tally Kernel**: Updates event counter tallies stored in global memory.
- **Event Kernel**: Processes next events for photons.

The Transport, Tally, and Event Kernels are repeated until all photons created by the Source Kernel have been terminated through either either absorption or escape. Since these CUDA kernels must be launched from the CPU, the number of terminated photons needs to be copied from the GPU back to the CPU after each iteration so that the CPU knows when to end the transport stage. When all particles have been terminated, both the history-based and event-based implementations need to copy the tally data from the GPU back to the CPU to communicate the final results to the user.

3.1.2 Particle Data Storage

Even though history-based and event-based transport algorithms call different CUDA kernels, they both use common code for creating photons, computing distance to surface and collision, identifying next event, moving the particle, and processing the next event. This common interface was achieved by defining a particle reference class that encapsulates where the particle data is actually stored. For the history-based implementation, the position, direction, weight, energy, and next event index of each photon is stored in fast register memory accessible only to the GPU thread associated with that photon. Using register memory is not possible for the event-based implementation because the photon data needs to persist between CUDA kernel launches. Therefore, all photon data must be stored in a particle bank in global memory. This increase in global memory transactions is one of the costs for switching to an event-based transport algorithm. However, the history-based transport algorithm may also need to use global memory when the amount of data stored per particle exceeds the capacity of register memory.

3.1.3 Random Number Generation

One of the core features needed by all Monte Carlo particle transport codes is a high quality method for generating random numbers. The most commonly used type of random number generator is the pseudorandom number generator, which produces a sequence of random numbers based on a fixed seed value. Pseudorandom number generators are essential for

reproducibility, which is important for verifying that the code consistently produces valid results. The quality of different pseudorandom number generators varies substantially, and it important to use one with a large period. The period determines how many random numbers can be generated before they start to repeat.

Although random numbers can be generated on the CPU and then copied to the GPU, this is not as efficient as generating random numbers directly on the GPU as needed. The pseudorandom number generator implemented in Savannah is a linear congruential generator defined by the recursion relation:

$$x_n = (ax_{n-1} + c) \bmod m, \tag{3.1}$$

where x_n is the state at step n, a is the multiplier, c is the increment, and m is the modulus. The state x_0 is called the seed value. This is the same random number generator used by WARP [7] and OpenMC [27] Monte Carlo neutron transport codes, with the former using single precision, and the latter using double precision. Since Savannah also uses double precision, the parameters in Equation 3.1 were set to the same values as the default in OpenMC: $a = 2806196910506780709$, $c = 1$, and $m = 2^{63}$ [28]. Using these values provides a period length of 2^{63} [29], which is good enough for a research code like Savannah, but generally not sufficient for a production code.

One of the advantages of using Equation 3.1 to generate random numbers is that the state can be stored separately from the recursion relation. This provides more flexibility for where to store the state, which means that the same random number generator can be used for both CPU and GPU execution spaces in Savannah. The GPU execution space stores a separate state for each particle in global memory, which is seeded directly on the GPU by using the 64-bit Sobol quasirandom number generator from NVIDIA's cuRAND library [30].

3.1.4 Events and Tallies

In addition to storing its own state for the random number generator, each photon created in Savannah is also assigned one of the following integer values indicating its next event type:

- **0**: Photon was created.
- **1**: Surface crossing for an uncollided photon.
- **2**: Surface crossing for a photon that has had at least one collision.
- **3**: Isotropic scattering collision interaction.
- **4**: Absorption collision interaction.
- **5**: Photon was terminated and no longer needs to be tracked.

Of these six event types, there are four that correspond to event counter tallies stored in global memory: uncollided escape, collided escape, number of scatter events, and number of absorption events. As these are all 64-bit integer tallies, they were implemented using the warp shuffle method discussed in Section 2.1 for both history-based and event-based transport algorithms. The key difference between the two different transport algorithms is that the history-based implementation tallies all the events for each particle in register memory before using the warp shuffle method to add the final results to global memory. This is currently feasible in Savannah due to the Big Kernel not using much register memory. As the tallies get bigger, however, they will not fit in register memory for the Big Kernel and a different tallying method will need to be used.

3.1.5 Particle Remapping

As was mentioned in Section 3.1.2, the increase in global memory transactions is one of the costs for switching to an event-based transport algorithm. Another cost is the concept of particle remapping, which is a process for sorting particles into event groups that will collectively execute the same set of instructions on the GPU. Only the particle index values are sorted, which is less expensive than moving all of the particle data. Particle remapping was introduced in WARP and implemented using a radix sort algorithm from the CUDPP library [7]. Although radix sort is an efficient sorting algorithm for integers, it was decided to implement the counting sort algorithm in Savannah instead. The counting sort algorithm is $O(n+k)$, where n is the number of particles and k is the number of event types. However, the biggest advantage of using counting sort is that the event counter tallies are already computed as a part of the process used to sort the particle index values. Therefore, the Tally Kernel only has to add these pre-computed event counts to the appropriate tallies stored in global memory after the remapping process is complete. This also removes the need to perform additional work to locate the boundaries between the event groups, such as is done in WARP [7].

To see how particle remapping works in Savannah, the process is represented graphically in Figure 3.1. Each photon is represented by a unique particle index number in a data array called the remap vector. Also associated with each photon is its next event type. After all the photons have been created in the Source Kernel, the remap vector consists of values that increase monotonically. In the Transport Kernel each photon is assigned its next event type, which could be either a surface crossing, scattering collision, or absorption collision. The remapping process occurs in the Remap Kernel[1] after the Transport Kernel, but before the Tally and Event Kernels. Once the Remap Kernel is complete, all the photons crossing a surface are listed first in the remap vector, all the photons that will scatter are listed next, and all the photons that will be absorbed are listed last. Now when the Event Kernel is launched the majority of warps will only include photons that need to execute instructions for the same event type.

[1]Counting sort algorithm actually requires four CUDA kernels to enable synchronization across blocks.

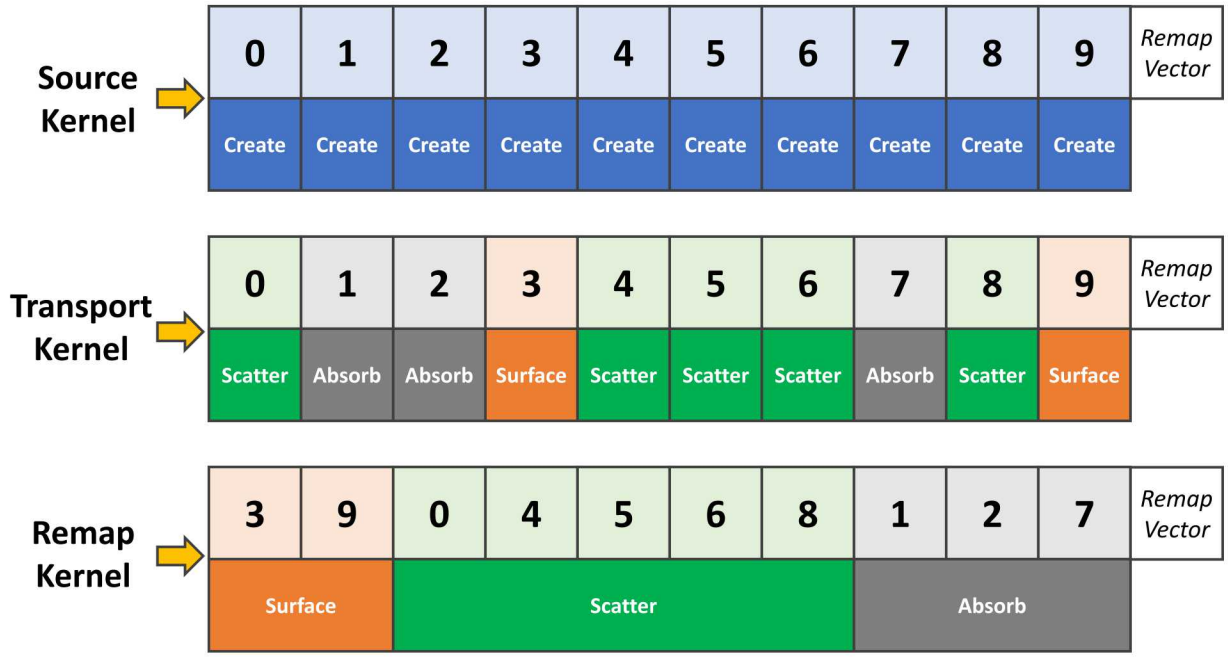

Figure 3.1: Particle remapping process for the event-based transport algorithm implemented in Savannah.

In addition to reducing branch divergence, the other advantage of particle remapping is that it moves all of the terminated particles to the end of the remap vector. This reduces the number of photons that are involved in future Transport and Event Kernel launches, as well as reducing the time it takes to remap the remaining active particles. Without remapping, the photons that are terminated are still included even though they do not contribute anything and all functions calls for terminated particles are null operations.

3.2 GPU Performance Testing

Two types of problems were used to test the performance of the history-based and event-based transport algorithms implemented in Savannah: photon attenuation and isotropic scattering. Both problem types were run on the three NVIDIA Tesla GPUs listed in Table 1.2, which are all available on different compute nodes on the Ride advanced architecture testbed at Sandia National Laboratories [31]. Unless otherwise stated, all timing data reported in the following two sections are an average of 10 independent runs of Savannah v0.2 that was compiled using GCC 5.4.0 and CUDA 8.0.44.

3.2.1 Photon Attenuation

The first problem type used to test the performance of history-based and event-based transport algorithms was photon attenuation. Mono-energetic photons were directed into a 1D slab made up of helium, with a linear attenuation coefficient of $\mu = 6.59936 \times 10^{-3}$ m^{-1}. The analytical solution for the fraction of photons that escape the problem domain is:

$$\frac{N_{\text{out}}}{N} = e^{-\mu x}, \tag{3.2}$$

where N_{out} is the number of photons that escape, N is the initial number of photons, and x is the thickness of the slab in meters. Expected results for three different test cases are summarized in Table 3.1.

Table 3.1: Different test cases used for tallying photon escape in a 1D helium slab.

Case	**Description**	$\boldsymbol{x}$ **(m)**	N_{out}/N
1	All the photons escape	0	1.0
2	Half of the photons escape	100	0.5
3	No photons escape	10,000	0.0

The test cases defined in Table 3.1 are identical to the ones previously used to test the performance of the different methods for tallying on the GPU [16]. In this work, however, all tallies use the warp shuffle method to avoid the substantial branch divergence that can occur when tallying via atomic functions in shared or global memory. Although some branch divergence is still possible, its impact should be much less noticeable. Table 3.2 shows the ratio of total runtimes of the event-based transport algorithm to the history-based transport algorithm for all three test cases. Results are reported with and without particle remapping.

Table 3.2: Ratio of total runtimes using event-based over history-based transport algorithms to solve a 1D photon attenuation problem with 10^8 histories.

	Remapping			**No Remapping**		
Case	*K40*	*K80*	*P100*	*K40*	*K80*	*P100*
1	3.959	4.187	1.963	2.038	2.114	1.171
2	4.090	4.241	2.190	2.082	2.142	1.194
3	4.019	4.258	2.044	2.060	2.136	1.184

As expected, Table 3.2 shows that there is not much variation in the results across the three different photon attenuation cases. The event-based transport algorithm with remapping was always about four times slower than the history-based implementation on the two Kepler GPUs, and about twice as slow on the P100. Without remapping, the event-based transport algorithm was only twice as slow on the Kepler GPUs, and about 20% slower on the P100. Given that photons only interact once, by either getting absorbed or escaping, there is therefore not enough branch divergence in these problems to offset the cost of remapping. To help understand where the cost of remapping in the event-based transport algorithm comes from, Figure 3.2 plots the combined runtimes of the initialize, transport, and finalize stages on the P100 for the test case where all photons escape.

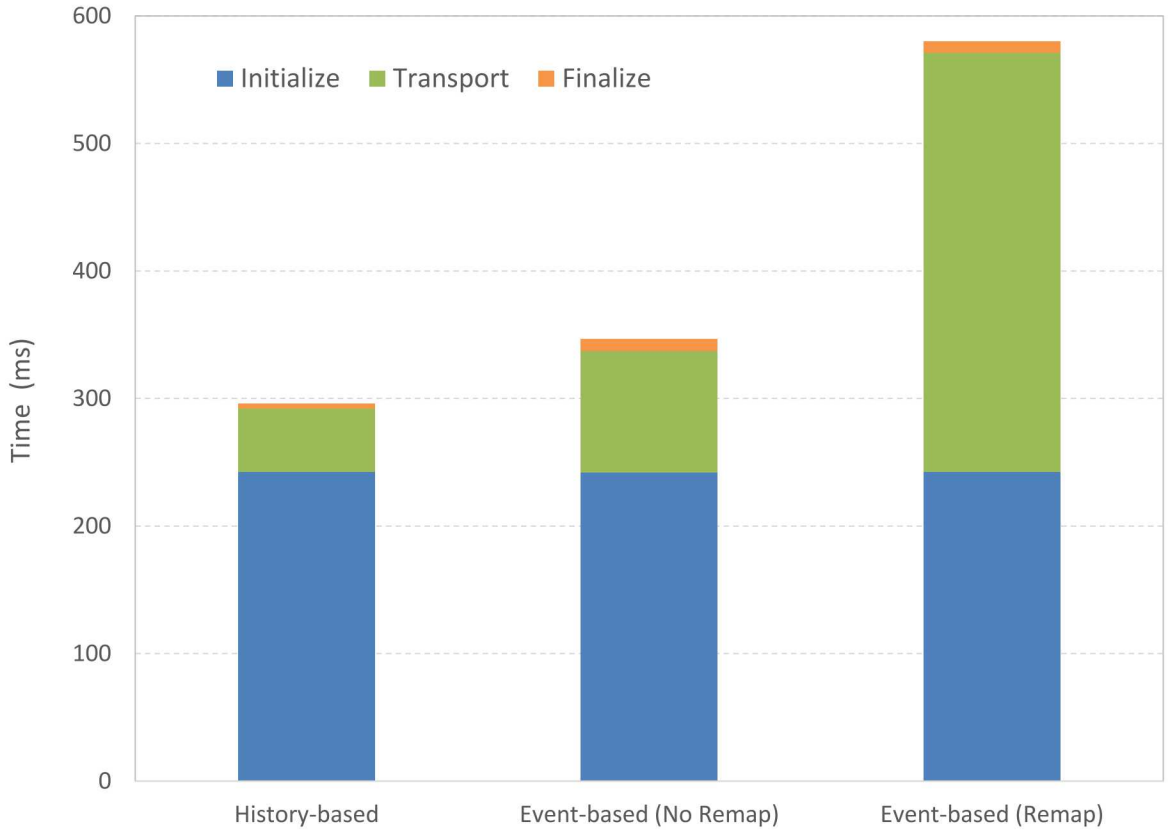

Figure 3.2: Breakdown of the total runtimes on the P100 for solving a 1D photon attenuation problem where all 10^8 photons escape.

Of the three stages plotted in Figure 3.2, only the transport stage shows a substantial difference between the three transport algorithms. Recall from Section 3.1.1 that the transport stage is reponsible for creating and processing particles on the GPU until all of the histories have been terminated. The runtimes for this transport stage are summarized in Table 3.3, as well as the portion of that runtime spent executing all the CUDA kernel

calls used by each transport algorithm.[2] Whereas the Big Kernel of the history-based transport algorithm only accounts for 50% of the time spent in the transport stage, all the CUDA kernels in the event-based transport algorithm require 80% when remapping is enabled. When remapping is disabled, only 40% of the time spent in the transport stage is used up by CUDA kernel calls. The breakdown of where time is spent in the event-based transport algorithm with and without remapping is summarized in Table 3.4. This data was obtained through the use of nvprof, NVIDIA's command-line profiling tool [32].

Table 3.3: Runtimes on the P100 for the transport stage and the sum of all its CUDA kernel calls when solving a 1D photon attenuation problem where all 10^8 photons escape.

Transport Algorithm Type	**Transport Stage (ms)**	**CUDA Kernels (ms)**
History-based	49.7	24.6
Event-based	95.7	40.4
Event-based with remapping	328.4	264.3

Table 3.4: Total runtime spent executing each CUDA kernel in an event-based transport algorithm when solving a 1D photon attenuation problem where all 10^8 photons escape.

Kernel Name	**Remapping (ms)**	**No Remapping (ms)**
Source Kernel	6.26	6.16
Transport Kernel	23.93	11.40
Tally Kernel	0.01	21.32
Event Kernel	13.94	1.49
Remap Kernel	220.16	N/A

Table 3.4 shows that while remapping had a negligible impact on the Source Kernel, it more than doubled the time spent in the Transport Kernel. This behavior can be explained by counting how many times the CUDA kernels were called. Without remapping, the Transport, Tally, and Event Kernels only need to be called once. With remapping, however, these three CUDA kernels and the Remap Kernel are called twice, even though all photons escape on the first iteration. This behavior occurs because of how the number of terminated particles are computed when remapping is enabled. The Transport Kernel assigns each photon with

[2]Excludes the CUDA kernel used by cuRAND to generate random numbers for all transport algorithms.

an event index of 1 to indicate that its next event will be a surface crossing. These events are then tallied in the Remap Kernel as part of the counting sort implementation, which is why the Tally Kernel is so efficient because it only needs to update the escape tally stored in global memory. After the remapping process is complete, the number of active particles remaining is copied from the CPU to the GPU. This value includes all particles that crossed a surface, which means that none of the photons will be terminated until after the next iteration. When remapping is disabled, the photons that escaped are included as part of the terminated particles. One improvement to the remapping implementation could therefore be to add a new event index for escaped particles to distinguish them from particles that crossed an inner surface. This distinction will become more important when there are multiple regions involved instead of a simple 1D slab geometry.

Although the increase in runtime spent in the Transport Kernel can be explained by the number of times it was called, this was not the case for the Event Kernel. Even accounting for calling the Event Kernel twice, the runtime with remapping enabled was almost five times longer than when remapping was disabled. Since all photons escape, there is no branch divergence in the Event Kernel with or without remapping. The only difference between the two event-based transport implementations is how the data in the particle bank is accessed. For optimal GPU performance, it is highly recommended that global memory read and write transactions are coalesced whenever possible [4]. This can be achieved by making adjacent threads in a warp access adjacent locations in memory at the same time, which can be managed as a single transaction instead of multiple transactions.

To measure how effectively a CUDA kernel uses global memory, there are two metrics available in nvprof called global load and store efficiency that define the ratio of requested global memory load or store throughput to the required global memory load or store throughput [32]. A value of 0% means that no memory transactions were coalesced, and a value of 100% indicates that all memory transactions were coalesced. When remapping is disabled, most read/write operations to global memory are coalesced in the Event Kernel, with global load and store efficiency values of 50% and 100% respectively. Values for the Event Kernel with remapping are much lower, as is shown in Table 3.5. This helps explain why the Event Kernel performs worse when remapping is enabled, even when identical code is being executed.

Table 3.5: Global load and store efficiency for the Event Kernel in an event-based transport algorithm with remapping.

Global Load Efficiency (%)			**Global Store Efficiency (%)**		
Minimum	*Maximum*	*Average*	*Minimum*	*Maximum*	*Average*
30.01	30.02	30.01	0.00	12.52	6.26

In addition to the cost of increasing non-coalesced memory transactions, using remapping in an event-based transport algorithm also comes with a significant cost for the remapping

process itself. Accounting for unnecessarily calling the Remap Kernel twice, the cost of remapping is still about 110 ms. Although this does include all the tallying, it is a non-trivial amount that effectively doubles the runtime compared to the case where remapping is not used. Therefore, for remapping to be effective, this cost must be offset by a significant amount of branch divergence.

3.2.2 Isotropic Scattering

In an effort to increase the amount of branch divergence, the second problem type used to test the performance of the history-based and event-based transport algorithms in Savannah was isotropic scattering. Mono-energetic photons were directed into a 1D slab with the artificial cross sections shown in Table 3.6 for absorption and isotropic scattering events. Artificial cross sections were used to highlight the impact on performance as the amount of scattering is increased, which ranges from no scattering and all photons getting absorbed, up to 100% scattering where photons can only scatter and all will eventually escape. If a photon gets absorbed, then in the Event Kernel it will have its direction, particle weight, and energy set to zero, as well as being marked as a terminated particle. If a photon scatters, then a new direction is computed based on isotropic scattering. The width of the 1D slab for all of these problems was fixed at 100 cm so that the full scattering case would not be too demanding on the total runtime.

Table 3.6: Artificial macroscopic cross sections used for tallying photon scattering and absorption events in a 100 cm slab.

Case	Scattering (cm^{-1})	Absorption (cm^{-1})
1	0.0	1.0
2	0.2	0.8
3	0.5	0.5
4	0.8	0.2
5	1.0	0.0

Total runtime for the five test cases defined in Table 3.6 are represented as a line chart in Figure 3.3 for the K40 and P100. The K80 results were on the same order of magnitude as the K40 results, which was expected given that they are similar architectures and only one of the two GPUs on the K80 was used to run the problems. All transport algorithms running on all GPU architectures take longer to run as the amount of scattering increases. This is expected as more iterations are needed for all the photons to get absorbed or escape, with the event-based transport algorithm requiring about 70 iterations when photons have an 80% chance of scattering, and over 45,000 iterations at 100% scattering.

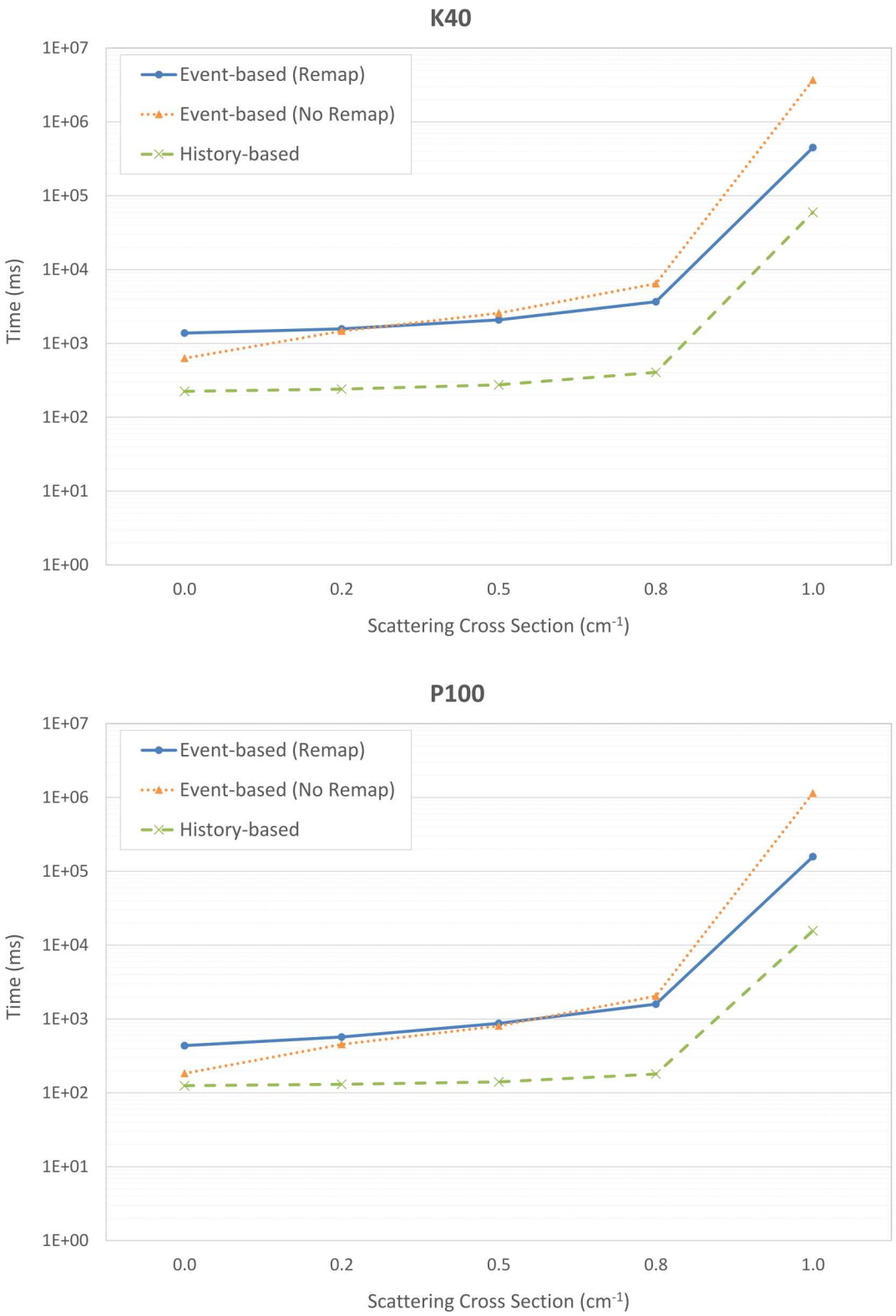

Figure 3.3: Impact on total runtime of history-based and event-based transport algorithms as isotropic scattering is increased for 10^8 histories.

Figure 3.3 also shows when remapping becomes important with respect to performance. The total runtime of the event-based transport algorithm with and without remapping on the K40 is similar when photons have a 20% chance of scattering, and on the P100 when photons have a 50% chance of scattering. As even more scattering occurs, the cost of disabling remapping starts to get much higher. This cost is captured in Table 3.7 as the difference in total runtimes with and without remapping. For low scattering cases, the difference in total runtime is only a fraction of a second. However, disabling remapping at 100% scattering costs about 16 minutes on the P100, and almost an hour on the K40 and K80.

Table 3.7: Cost of disabling remapping in an event-based transport algorithm to solve an isotropic scattering problem with 10^8 histories.

	Difference in Total Runtime (s)		
Case	*K40*	*K80*	*P100*
1	−0.75	−0.67	−0.25
2	−0.10	0.02	−0.12
3	0.49	0.70	−0.06
4	2.78	3.36	0.45
5	3229.80	3341.18	985.70

The cost of disabling remapping in the event-based transport algorithm is expected to get worse as more branch divergence occurs in the simulation, which will likely impact the Transport and Event Kernels the most. Figure 3.4 is a line chart showing the duration of sequential calls for these two CUDA kernels when photons have a 50% chance of scattering. Note that the first call for the Transport Kernel has the same duration with and without remapping, but the Event Kernel takes longer when remapping is enabled. This behavior occurs because the Transport Kernel is called before the Remap Kernel, but the Event Kernel is called after the Remap Kernel. Remapping the particle indices changes the global memory access pattern, as was observed with the photon attenuation problem in Section 3.2.1. Changing the global memory access pattern is also the most likely reason as to why the duration of the Transport Kernel increases with the second call, since it now is called after particle indices have been remapped.

Although enabling remapping results in a greater initial cost for the Transport and Event Kernels, this cost is steadily reduced with subsequent calls. When remapping has been disabled, however, the duration of both these CUDA kernels approaches a fixed value. The explanation for this behavior lies in the number of photons that are included in each iteration. With remapping, all the terminated particles are moved to the end of the remap vector because they have the highest event index number, which allows them to be ignored in future iterations. This reduces the number of active particles in not only the Transport and Event

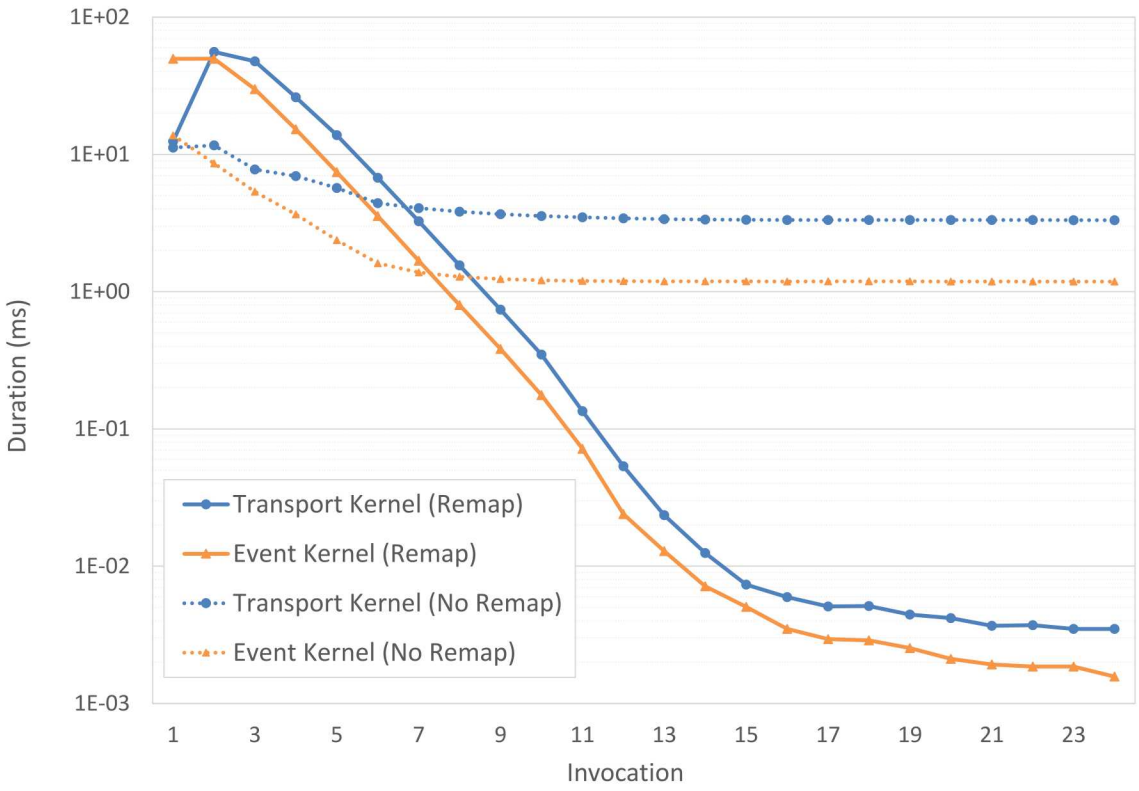

Figure 3.4: Duration of sequential CUDA kernel calls on the P100 for the case where each interaction for 10^8 histories has a 50% chance of scattering.

Kernels, but also the Remap Kernel. The cost of remapping is therefore significantly reduced as more particles become inactive. In comparison, when remapping has been disabled the terminated particles are scattered throughout the particle bank and cannot be ignored. Although these terminated particles do not contribute much to the simulation, there is still some overhead involved in including them.

Figure 3.5 shows how the percentage of time spent in each CUDA kernel changes as the amount of scattering is increased. When remapping is enabled, the largest cost with no scattering is clearly the Remap Kernel, which is consistent with the photon attenuation results from Section 3.2.1. At 100% scattering the dominant cost becomes the Transport Kernel as it gets signficantly cheaper to remap when there are fewer active particles to sort. In contrast, the largest cost when remapping is disabled is always the Tally Kernel. This cost increases substantially for the 100% scattering case, most likely because the warp shuffle method used for tallying is being performed by both active and inactive particles. Even though improving the Tally Kernel could have a substantial impact on the performance, the cost of the Transport Kernel on its own is higher at 100% scattering than the total runtime when remapping is enabled.

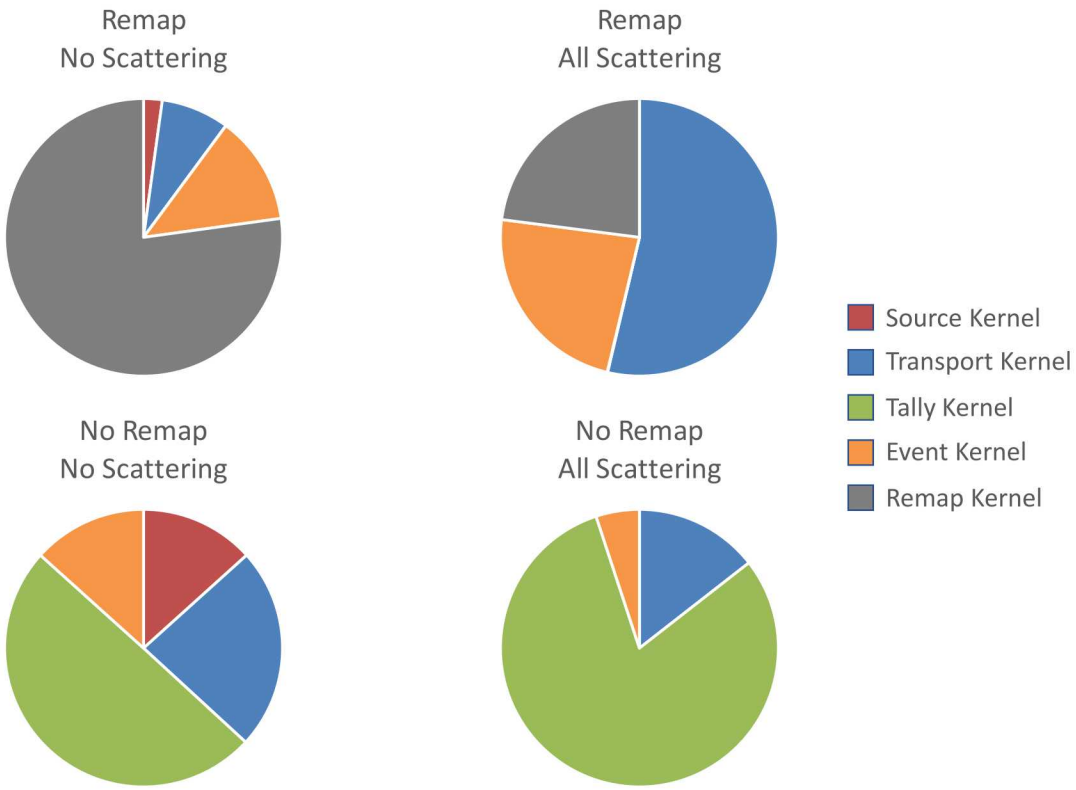

Figure 3.5: Percentage of time spent on the P100 in each CUDA kernel of the event-based transport algorithm for no scattering and 100% scattering cases.

Results for the isotropic scattering test cases with and without remapping have shown that it is generally better to use remapping, especially for long-running simulations. Referring back to Figure 3.3, however, note that the history-based transport algorithm always outperformed the event-based implementation whether or not remapping was enabled. Table 3.8 shows the ratio of total runtimes using event-based over history-based transport algorithms for all test cases and GPU architectures considered. The event-based transport algorithm with remapping enabled was 3.5 to 10 times slower, which is consistent with the factor of 3 to 7 reported in the work done at Oak Ridge National Laboratory [11]. In comparison, the event-based transport algorithm without remapping ranged from as low as 1.5 times slower for no scattering, up to 73 times slower for 100% scattering on the P100.

To understand more about how history-based and event-based transport algorithms behave as scattering increases, Figure 3.6 on page 47 shows a line chart plotting the total time spent in some of the key CUDA kernels. Not shown are the Source Kernel, which was always 6.3 ms, and the Tally Kernel, which is negligible due to all the actual tallying being done in the Remap Kernel. With no scattering, the Transport Kernel of the event-based transport algorithm takes about the same amount of time as the entire history-based transport algorithm implemented in the Big Kernel. As more scattering is introduced, the cost of the Transport Kernel increases faster than the Big Kernel. The most likely

Table 3.8: Ratio of total runtimes using event-based over history-based transport algorithms to solve an isotropic scattering problem with 10^8 histories.

	Remapping			**No Remapping**		
Case	*K40*	*K80*	*P100*	*K40*	*K80*	*P100*
1	6.146	4.944	3.489	2.805	2.336	1.466
2	6.528	5.633	4.370	6.096	5.706	3.459
3	7.558	6.453	6.199	9.357	8.853	5.748
4	9.009	7.331	8.860	15.853	14.813	11.370
5	7.554	6.556	10.064	61.818	62.228	72.934

explanation for this behavior is due to the particle data being stored in global memory instead of registers, even though the same code is used to process each particle. The Big Kernel performed 25 million global load and store transactions, whereas the Transport Kernel performed up to 324 million global load transactions, and up to 62 million global store transactions.

Even though the Transport, Event, and Remap Kernels all took longer to process than the Big Kernel, the primary reason for considering the event-based transport algorithm is to reduce branch divergence. Figure 3.7 on page 48 shows the average warp execution efficiency for the Big Kernel compared to the Transport and Event Kernels for all five test cases considered. Warp execution efficiency measures the amount of branch divergence in a CUDA kernel, and is defined as the ratio of the average active threads per warp to the maximum number of threads per warp supported on an SM. The Big Kernel in the history-based transport algorithm is highly sensitive to branch divergence, with the warp execution efficiency dropping to below 10% at 100% scattering. In contrast, the Transport and Event Kernels in the event-based transport algorithm seem to approach fixed values of around 80% and 90% no matter how much scattering occurs. These results show that the event-based transport algorithm is achieving the goal it was originally set out to achieve, namely reducing branch divergence. However, for the event-based transport algorithm to be able to outperform the history-based implementation, the time spent in each divergent path would need to be high enough to offset the increased global memory transactions and cost of remapping particles. In Savannah, the divergent paths only perform at most 10 operations, which could help explain why the event-based transport algorithm does not perform as well as the history-based implementation in these tests. In a production Monte Carlo particle transport code, it is expected that there will be much longer divergent paths, especially for coupled electron-photon radiation transport.

In addition to observing how the amount of scattering impacts the total runtime of the history-based and event-based transport algorithms, timing data was also obtained on the P100 with a varying number of particle histories. The results for a low and high scattering

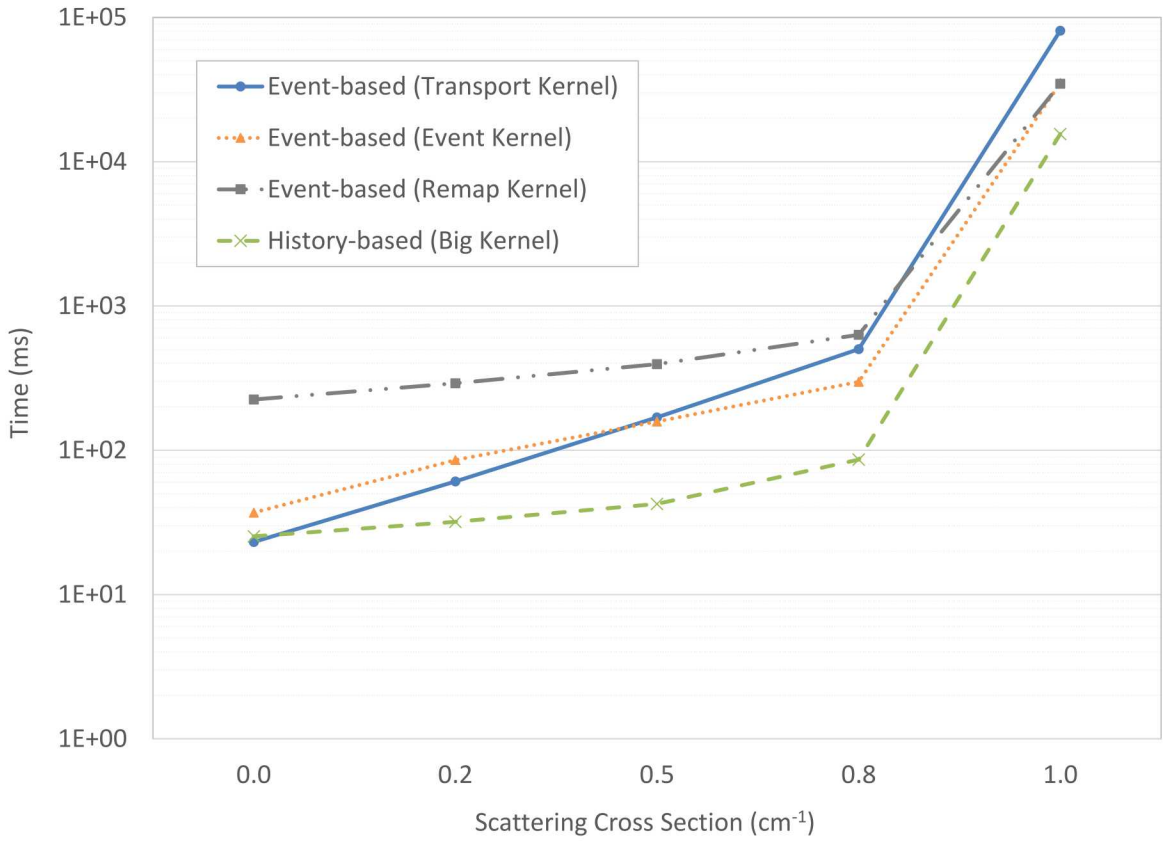

Figure 3.6: P100 timing data for key CUDA kernels in the history-based and event-based (with remapping) transport algorithms as isotropic scattering is increased for 10^8 histories.

test case are plotted in the log-log graph shown in Figure 3.8 on page 49. Note that these are the only results presented in this work that were obtained by running Savannah compiled with CUDA 9, which was required due to system changes on Ride. Trendlines fitted to the data in Figure 3.8 show that there is a strong linear relationship between total runtime and the number of histories. This linear relationship is summarized in Table 3.9. Both transport algorithms have a fixed cost of approximately 100 ms, but the cost per particle for the event-based transport algorithm was up to two orders of magnitude higher. Reducing the number of histories can therefore improve the overall performance of the event-based transport algorithm compared to the history-based implementation. For example, using 10^7 histories instead of 10^8 histories reduces the total runtime from being eight times slower to only being twice as slow.

The improvement in performance of the event-based transport algorithm compared to the history-based transport algorithm with fewer particles does not seem to be caused by a difference in the total runtimes of the CUDA kernels. All CUDA kernels in the history-based and event-based transport algorithms dependent on the number of histories are reduced by

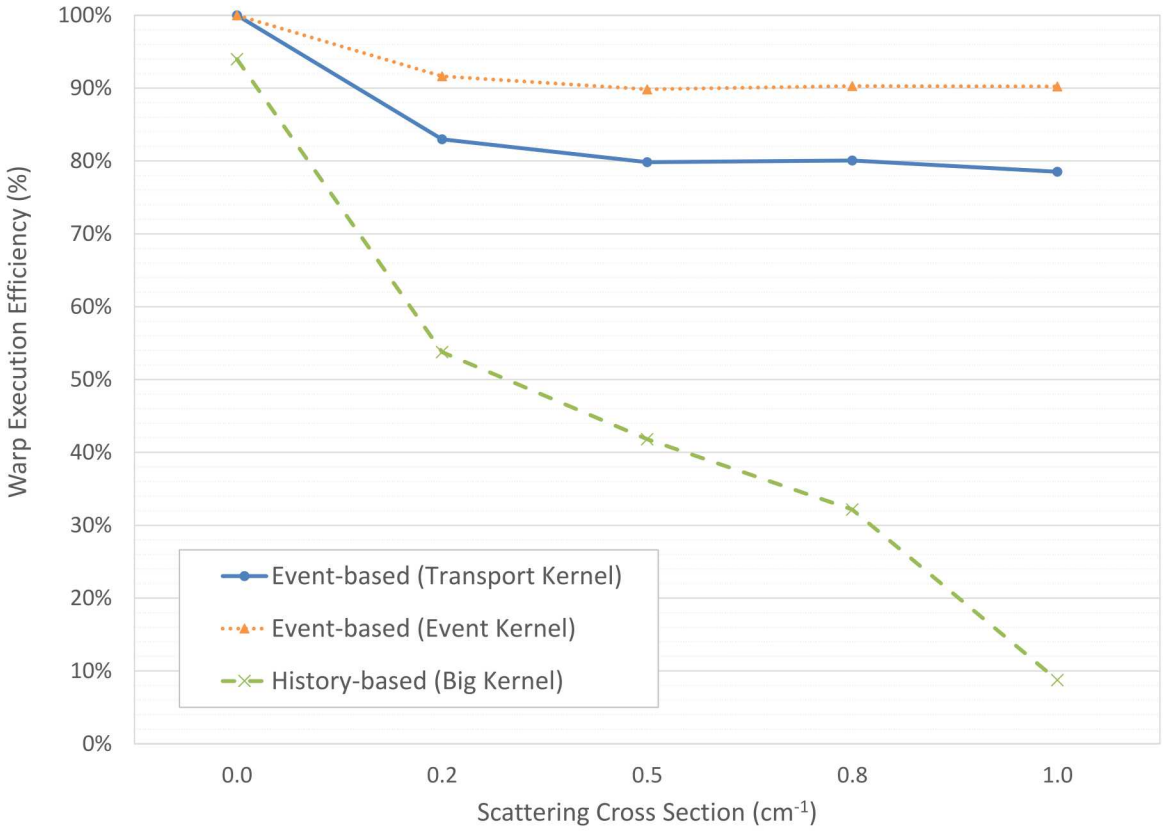

Figure 3.7: Average warp execution efficiency on the P100 for key CUDA kernels in the history-based and event-based (with remapping) transport algorithms.

a factor of 10 when using 10^7 histories instead of 10^8 histories. However, there is a large difference in how much of the runtime in the transport stage is spent executing these CUDA kernels. For the high scattering case, the Big Kernel accounts for 87% of the transport stage with 10^8 histories, but drops to only 35% with 10^7 histories. In contrast, the cost of all the CUDA kernels in the event-based transport algorithm account for 96% and 83% respectively. These results indicate that the difference is caused by actions performed on the CPU, not the GPU. Further research is required to identify what instructions on the CPU are responsible for this behavior.

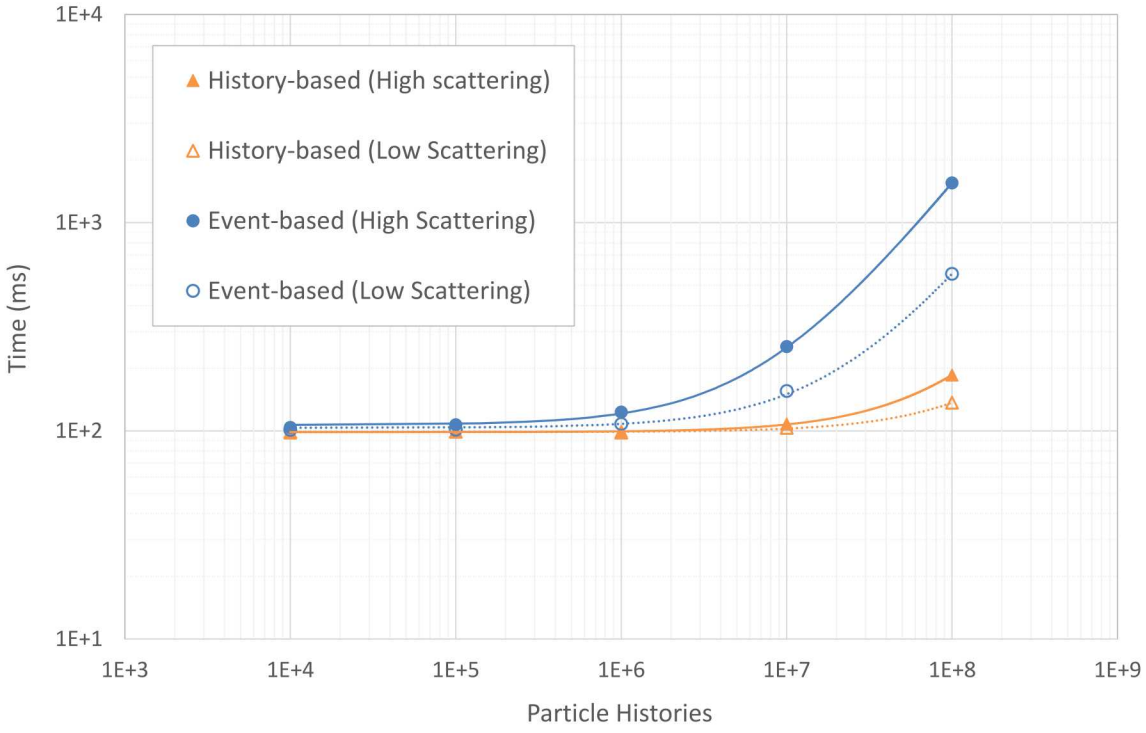

Figure 3.8: Total runtime on the P100 versus number of particle histories for history-based and event-based transport algorithms. Low and high scattering cases refer to a 20% and 80% chance to scatter per interaction respectively.

Table 3.9: Parameters describing the linear relationship between total runtime and particle histories for timing data obtained on the P100 for an isotropic scattering problem with 10^8 histories.

History-based			
Case	*Cost Per Particle (ms)*	*Fixed Cost (ms)*	R^2 *Fit*
Low Scattering (20%)	3.8×10^{-7}	98.7	0.9984064
High Scattering (80%)	8.7×10^{-7}	98.6	0.9997208
Event-based with Remapping			
Case	*Cost Per Particle (ms)*	*Fixed Cost (ms)*	R^2 *Fit*
Low Scattering (20%)	4.6×10^{-6}	103.4	0.9997304
High Scattering (80%)	1.4×10^{-5}	106.7	0.9999878

This page intentionally left blank.

Chapter 4

Conclusions and Future Work

Next-generation architectures are trending towards using accelerators like GPUs to boost compute power, which is problematic for Monte Carlo particle transport codes because of their high memory usage and divergent algorithms. As a result, codes such as ITS at Sandia National Laboratories must learn to evolve from methods that have been relied on for decades. Two key issues were explored in this report, namely how to implement memory-intensive tallies, and also how to reduce divergence using an event-based transport algorithm. This concluding chapter summarizes lessons learned, and also discusses future work that is still needed to improve the performance of production Monte Carlo particle transport codes on GPUs.

4.1 Monte Carlo Tallies on the GPU

The most commonly used method for processing Monte Carlo tallies in parallel is to use tally replication. While tally replication is very effective for CPU-based systems, simply copying the tally structure to every thread is not always feasible for the GPU. Memory resources are more limited on the GPU than the CPU, and these resources must be shared by not only all the tallies, but also other memory-intensive data such as the geometry and cross sections.

For integer-based event counters with a high update frequency, the most effective alternative to tally replication is to use the warp shuffle method. The warp shuffle method allows threads in a warp to share data so that an efficient parallel reduction can be performed. For event counters or other tallies that need to use floating-point values, an even better solution is to use the block reduction method. The block reduction method only requires one atomic function call per block, which are generally more expensive for floating-point values.

For memory-intensive particle flux mesh tallies, performance results presented in this report have shown that the KDE integral-track and stratified sampling implementations can both be efficiently processed on the GPU by assigning threads to mesh nodes or elements instead of particle tracks. Assigning GPU threads to mesh nodes or elements allows each thread to only access its own tally structure, which removes the need for using expensive atomic function calls or tally replication to avoid race conditions.

Even though the alternative tally implementations discussed in this report may allow tallies to be processed on the GPU during the Monte Carlo particle transport simulation, future efforts should be made towards exploring the use of GPUs as tally servers [33]. If GPUs were used as tally servers, then most of the available memory could be dedicated to the tallies, which may make tally replication a more feasible option. Tally servers also would make it easier to allow both in-situ tallying and post-processing tallying within the same framework.

4.2 Event-Based Monte Carlo Particle Transport

A new research code called Savannah was also introduced in this work, which was designed to explore the effectiveness of using an event-based transport algorithm. Performance results using Savannah were presented for two types of problems: photon attenuation and isotropic scattering. The history-based transport algorithm actually outperformed the event-based implementation for all the test cases considered, with speedups ranging from 20% up to a factor of 10 times faster. This was not expected, given that the event-based transport algorithm did substantially reduce divergence, and should therefore be better suited to execution on the GPU.

Although the event-based transport algorithm did not perform as expected, a closer look at the results was able to highlight the costs that are involved in switching to this alternative algorithm. The most obvious cost is the time needed to remap particles. For low scattering problems, remapping was clearly the most time-consuming step performed by the event-based transport algorithm. For high scattering problems, however, the cost of remapping had a much lower impact. The reason for this behavior was because remapping can ignore terminated particles, which makes it faster to process the remaining active particles.

A more subtle cost of the event-based transport algorithm was discovered by comparing performance results with and without remapping. For both photon attenuation and isotropic scattering test cases, the results showed that there is a noticeable increase in runtime caused by the way the algorithm accesses the particle bank stored in global memory. When remapping was enabled, the number of global memory transactions that were coalesced was substantially reduced, especially when updating the data in the particle bank. This increases the total number of global memory transactions that are needed to produce the same results. For high scattering cases, however, disabling remapping ends up costing much more than the increase in global memory transactions.

While the event-based transport algorithm did not perform as well as expected, this will likely change as more complexity is added to the history-based implementation. The divergent paths implemented in Savannah involved fewer than 10 instructions, which is a small number compared to how many instructions would be needed to solve real problems in a coupled electron-photon radiation transport code like ITS. Future work could therefore study the impact of increasing the time spent in divergent paths, perhaps by including more complicated physics for electrons and photons.

References

[1] TOP500, "Top 500 Supercomputing Sites: June 2018," https://www.top500.org/lists/2018/06/ (2018).

[2] J. A. HALBLEIB and T. A. MEHLHORN, "ITS: The Integrated TIGER Series of Coupled Electron/Photon Monte Carlo Transport Codes," *Nuclear Science and Engineering*, **92**, p. 338 (1986).

[3] NVIDIA CORPORATION, "CUDA C Programming Guide," PG-02829-001_v8.0 (2017).

[4] NVIDIA CORPORATION, "CUDA Best Practices Guide," DG-05603-001_v8.0 (2017).

[5] J. LUITJENS, *Faster Parallel Reductions on Kepler*, NVIDIA Corporation, https://devblogs.nvidia.com/faster-parallel-reductions-kepler (2014).

[6] F. B. BROWN and W. R. MARTIN, "Monte Carlo Methods for Radiation Transport Analysis on Vector Computers," *Progress in Nuclear Energy*, **14**, *3*, pp. 269–299 (1984).

[7] R. M. BERGMANN and J. L. 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**, pp. 176–193 (2015).

[8] R. C. BLEILE, P. S. BRANTLEY, S. A. DAWSON, M. J. O'BRIEN, and H. CHILDS, "Investigation of Portable Event Based Monte Carlo Transport Using the Nvidia Thrust Library," *Transactions of the American Nuclear Society*, **114**, pp. 369–372 (2016).

[9] D. F. RICHARDS, R. C. BLEILE, P. S. BRANTLEY, S. A. DAWSON, M. S. MCKINLEY, and M. J. O'BRIEN, "Quicksilver: A Proxy App for the Monte Carlo Transport Code Mercury," *Proceedings of 2017 IEEE International Conference on Cluster Computing*, Honolulu, Hawaii, September 5-8 (2017).

[10] M. MARTINEAU and S. MCINTOSH-SMITH, "Exploring on-node parallelism with neutral, a Monte Carlo neutral particle transport mini-app," *Proceedings of 2017 IEEE International Conference on Cluster Computing*, Honolulu, Hawaii, September 5-8 (2017).

[11] S. P. HAMILTON, S. R. SLATTERY, and T. M. EVANS, "Multigroup Monte Carlo on GPUs: Comparison of history- and event-based algorithms," *Annals of Nuclear Energy*, **113**, pp. 506–518 (2018).

[12] X. G. XU, T. LIU, L. SU, X. DU, M. RIBLETT, W. JI, D. GU, C. D. CAROTHERS, M. S. SHEPHARD, F. B. BROWN, M. K. KALRA, and B. LIU, "ARCHER, a

new Monte Carlo software tool for emerging heterogeneous computing environments," *Annals of Nuclear Energy*, **82**, pp. 2–9 (2015).

[13] R. C. BLEILE, P. S. BRANTLEY, M. J. O'BRIEN, and H. CHILDS, "Algorithmic Improvements for Portable Event-Based Monte Carlo Transport Using the Nvidia Thrust Library," *Transactions of the American Nuclear Society*, **115**, pp. 535–538 (2016).

[14] S. P. HAMILTON, T. M. EVANS, and S. R. SLATTERY, "GPU Acceleration of History-Based Multigroup Monte Carlo," *Transactions of the American Nuclear Society*, **115**, pp. 527–530 (2016).

[15] K. L. DUNN and P. P. H. WILSON, "Monte Carlo Mesh Tallies based on a Kernel Density Estimator Approach using Integrated Particle Tracks," *Proceedings of International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering (M&C 2013)*, Sun Valley, Idaho, May 5-9 (2013).

[16] K. L. BOSSLER, "Methods for Computing Monte Carlo Tallies on the GPU," *Proceedings of the Physics of Reactors Conference (PHYSOR 2018)*, Cancun, Mexico, April 22-26 (2018).

[17] K. L. BOSSLER, "Performance of Kernel Density Estimated Mesh Tallies on GPUs," *Proceedings of International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering (M&C 2017)*, Jeju, Korea, April 16-20 (2017).

[18] L. SU, Y. YANG, B. BEDNARZ, E. STERPIN, X. DU, T. LIU, W. JI, and X. G. XU, "ARCHER$_{\mathrm{RT}}$ – A GPU-based and photon-electron coupled Monte Carlo dose computing engine for radiation therapy Software development and application to helical tomotherapy," *Medical Physics*, **41**, *7*, pp. 1–13 (2014).

[19] S. HISSOINY, B. OZELL, H. BOUCHARD, and P. DESPRÉS, "GPUMCD: A new GPU-oriented Monte Carlo dose calculation platform," *Medical Physics*, **38**, *2*, pp. 754–764 (2011).

[20] X. JIA, X. GU, J. SEMPAU, D. CHOI, A. MAJUMDAR, and S. B. JIANG, "Development of a GPU-based Monte Carlo dose calculation code for coupled electron-photon transport," *Physics in Medicine and Biology*, **55**, pp. 3077–3086 (2010).

[21] X. WENG, Y. YAN, H. SHU, J. WANG, S. B. JIANG, and L. LUO, "A vectorized Monte Carlo code for radiotherapy treatment planning dose calculation," *Physics in Medicine and Biology*, **48**, pp. 111–120 (2003).

[22] X-5 MONTE CARLO TEAM, *MCNP - A General Monte Carlo N-Particle Transport Code, Version 5. Volume I: Overview and Theory.*, Los Alamos National Laboratory, Los Alamos, NM (2003).

[23] J. LEPPÄNEN, *Development of a New Monte Carlo Reactor Physics Code*, Ph.D. thesis, Helsinki Institute of Technology (2007).

[24] B. MOLNAR, G. TOLNAI, D. LEGRADY, and M. SZIEBERTH, "Vectorized Monte Carlo for Guardyan - a GPU Accelerated Reactor Dynamics Code," *Proceedings of the Physics of Reactors Conference (PHYSOR 2018)*, Cancun, Mexico, April 22-26 (2018).

[25] S. P. HAMILTON, T. M. EVANS, and S. R. SLATTERY, "Continuous-Energy Monte Carlo Neutron Transport on GPUs in Shift," *Transactions of the American Nuclear Society*, **118**, pp. 401–403 (2018).

[26] H. C. EDWARDS, C. R. TROTT, and D. SUNDERLAND, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, **74**, *12*, pp. 3202–3216 (2014).

[27] P. K. ROMANO, N. E. HORELIK, B. R. HERMAN, A. G. NELSON, B. FORGET, and K. SMITH, "OpenMC: A State-of-the-Art Monte Carlo Code for Research and Development," *Annals of Nuclear Energy*, **82**, pp. 90–97 (2015).

[28] MASSACHUSETTS INSTITUTE OF TECHNOLOGY, "The OpenMC Monte Carlo Code (v0.10.0)," https://openmc.readthedocs.io/en/stable/index.html (2018).

[29] P. L'ECUYER, "Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure," *Mathematics of Computation*, **68**, *225*, pp. 249–260 (1999).

[30] NVIDIA CORPORATION, "cuRAND Library," PG-05328-050_v8.0 (2016).

[31] SANDIA NATIONAL LABORATORIES, *Advanced Systems Technology Test Beds*, National Technology and Engineering Solutions of Sandia, LLC, https://www.sandia.gov/asc/computational_systems/HAAPS.html (2018).

[32] NVIDIA CORPORATION, "Profiler User's Guide," DU-05982-001_v8.0 (2017).

[33] P. K. ROMANO, A. R. SIEGEL, B. FORGET, and K. SMITH, "Data decomposition of Monte Carlo particle transport simulations via tally servers," *Journal of Computational Physics*, **252**, pp. 20–36 (2013).

This page intentionally left blank.

Glossary

GPU Computing Terms

accelerator
Specialized hardware such as a GPU that is added to a computing system to help boost overall performance.

architecture
Defines how a computer or one of its components is designed with respect to both software and/or hardware.

atomic function
A function that performs a read-modify-write operation on data stored in either shared memory or global memory. This operation is performed independently of all other threads to avoid race conditions.

block synchronization
Process of synchronizing all the threads in a block before executing more instructions that depend on a previous result. This is often required when using shared memory to allow communication between the threads.

branch divergence
Occurs when threads in the same warp need to execute different instructions, which increases the total number of instructions executed for the whole warp.

cached memory
Temporary storage of data closer to the processing unit, which allows future access requests for that data to be more efficient than accessing its original location.

coalesced memory transaction
Occurs when adjacent threads in a warp access adjacent locations in global memory. Coalesced access to global memory is performed collectively as one transaction, rather than individually over multiple transactions.

constant memory
Small read-only memory space available to all threads on a GPU that is cached on-chip. Most efficient when all threads in a warp access a few distinct locations.

cuRAND
A library included in the CUDA toolkit that provides support for pseudorandom and quasirandom number generation on the GPU.

CUDA
: A parallel computing platform and programming model created by NVIDIA to enable general purpose computing on their GPUs.

CUDA compute capability
: Version number of an NVIDIA GPU (i.e., X.Y), which identifies its core architecture type and what CUDA features are supported.

CUDA core
: Single processing unit on an SM.

CUDA kernel
: A function launched from the host that is executed k times in parallel by k different threads on the device.

CUDA toolkit
: A development environment for creating NVIDIA GPU-accelerated applications.

device
: Refers to the GPU responsible for accelerating parts of the application.

execution space
: An abstraction that determines where an application will be run, either on the CPU or a GPU.

global load efficiency
: Ratio of requested throughput to required throughput for reading data from global memory. Measures amount of coalesced read transactions in a CUDA kernel.

global memory
: Largest memory space available to all threads on a GPU, but also has the slowest access rates because it is located off-chip.

global store efficiency
: Ratio of requested throughput to required throughput for writing data to global memory. Measures amount of coalesced write transactions in a CUDA kernel.

host
: Refers to the CPU responsible for running the application.

Kokkos
: A programming model for writing performance portable applications across diverse manycore processors, which includes multi-core CPUs and GPUs.

local memory
: Off-chip memory available to individual threads that is used when a thread requires more registers than are available to execute a CUDA kernel.

off-chip
: Components not built on the same integrated circuit as the processing unit.

on-chip
: Components built on the same integrated circuit as the processing unit.

parallel processing
: Multiple instructions and/or data are executed simultaneously. GPUs execute a single instruction on multiple data, making them well-suited to data-parallel algorithms.

race condition
: Two or more threads attempt to change a shared resource (i.e., memory location) at the same time, causing unexpected results.

reduction
: A parallel pattern that is used to combine all values in an array into one value, using an operator such as addition.

register memory
: Fastest on-chip memory available to individual threads, but is also a limited resource that is distributed among all the threads assigned to an SM.

serial processing
: Instructions of a program are executed one at a time in a sequential order.

shared memory
: Small on-chip memory space available to all threads in one thread block, which enables efficient communication between threads.

speedup
: Measures the relative performance of two different implementations used to process the same problem.

streaming multiprocessor (SM)
: Component of a GPU with many CUDA cores that manages, schedules, and executes warps. A GPU has multiple SMs.

texture memory
: Read-only memory available to all threads on a GPU that is cached on-chip. Most efficient when all threads in a warp read from locations that are close to one another.

thread
: Smallest programmable unit that executes instructions on a GPU.

thread block
: A programming abstraction that represents a group of warps.

vector processor
: A CPU that is able to process a single instruction on multiple data. In comparison, a scalar processor can only process a single instruction on single data.

warp
: Group of threads that execute common instructions in parallel on a GPU.

warp execution efficiency
: Percentage of active threads per warp on average being processed by an SM. Measures branch divergence of a CUDA kernel.

warp shuffle
: CUDA feature that allows threads in a warp to simultaneously exchange or broadcast data without needing to use shared memory.

Monte Carlo Particle Transport Terms

absorption
: Collision interaction where the particle and its energy is absorbed by the background material with which it interacts.

active particle
: Particle that is still being tracked by the transport algorithm.

collision interaction
: Type of event where the particle interacts with the background material and changes its direction and/or energy.

cross section
: Defines the probability that a particle experiences a given type of collision interaction (i.e., scattering or absorption). Usually measured in units of area, but can also be represented in units of per-unit-length if multiplied by the atomic number density of the background material.

estimator
: A mathematical rule that calculates an estimate of a quantity of interest based on observed data (e.g., average number of collisions in a region).

event
: Something that can happen to a particle. Examples include creation, absorption, scatter, move, surface crossing, and escape.

event-based transport algorithm
: An alternative Monte Carlo particle transport algorithm where particles are processed in groups according to the type of event that they are expected to experience next.

event counter
: A simple tally used to count the number of occurrences of a single event type.

history
: A sequence of events that one particle experiences until it gets absorbed or otherwise escapes the system.

history-based transport algorithm
: The traditional Monte Carlo particle transport algorithm where individual particles are followed from their birth until their death.

inactive particle
: Particle that has been terminated due to either getting absorbed or having escaped the problem domain.

isotropic scattering
: Scattering interaction where the new direction of the particle is selected by uniformly sampling a value from 0 to 360 degrees (i.e., all directions are equally likely to occur).

kernel density estimator (KDE)
: Statistical method used to estimate an unknown probability density function based on a fixed set of observations randomly sampled from that function.

KDE integral-track mesh tally
: Particle flux tally that uses the kernel density estimator to approximate particle flux at all the nodes of a mesh. Alternative to conventional histogram-based tallies that only compute average particle flux values for each mesh element.

mesh
: Discretization of a spatial domain into elements. Mesh elements can be any shape, including triangle or quadrilateral in 2D, and tetrahedron or hexahedron in 3D.

mesh node
: A point (i.e., vertex) on a mesh element where two or more edges meet.

mesh tally
: A tally that accumulates scores for a physical quantity of interest on all the elements or nodes of a mesh.

Monte Carlo particle transport
: Algorithm that uses random sampling to determine the movement and interactions of particles within one or more background materials.

nodal coordinates
: The (x, y, z) coordinates of one node on a mesh element.

particle
: Refers to one physical particle (e.g., electron, photon, neutron).

particle escape tally
: Tally that counts the number of particles that escape a region of interest.

particle flux tally
: Tally that measures the total length traveled by all particles in a region of interest per unit volume and time.

particle track
: Distance a particle travels along its current trajectory.

photon attenuation
: Gradual reduction in the intensity of a beam of photons emitted from a source as the photons travel through a material. Loss is due to collision interactions, not escape.

point-in-element search
: Algorithm to locate the mesh element in which a point defined by its (x, y, z) coordinates is located.

probability density function
: A mathematical function $f(x)$ of a continuous random variable X, whose integral from a to b defines the probability that the value of X falls within a and b.

random number generator
: Algorithm used to generate random numbers that determine when and how particles interact in a Monte Carlo particle transport simulation.

remap vector
: Sorted list of particle indices grouped together by next event type. All particles with event type 0 are listed first, then all particles with event type 1 are listed second, and so on until all active particles are included.

remapping
: Process used in the event-based transport algorithm to sort particles into event groups without physically moving all their data.

scattering
: Collision interaction where the particle is deflected from its original direction. May also result in a change of energy.

slab geometry
: 1D geometrical representation defined by a minimum and maximum boundary.

source
: Location where particles get created, which determines parameters such as their initial type, direction, and energy.

strata point
: A point on a subtrack defined by its (x, y, z) coordinates that was chosen at random.

stratified sampling mesh tally
: Particle flux tally that approximates the average particle flux in each element of a mesh. Uses a stochastic approach called stratified sampling to choose strata points, then apportions particle tracks based on where those points are located.

subtrack
: One section of a particle track after that particle track has been divided into k pieces. The length of each subtrack is equal to the total particle track length divided by k.

surface crossing
: Type of event where the particle crosses a geometrical surface, usually without changing its direction or energy.

tally
: Accumulates scores for a physical quantity of interest in one or more regions of interest. Tallies are often subdivided into different bins for energy, angle, and time.

tally replication
: Process of copying the tally bin structure onto multiple compute nodes or GPU threads. Final results are obtained by accumulating results from all the copies of the tally.

tally score
: A value computed for one or more events in a history using an estimator.

DISTRIBUTION:

1 Frank Angers
University of Michigan
500 S State St.
Ann Arbor, MI 48109

1	MS 1179	Martin Crawford, 1341
1	MS 1179	Ronald P. Kensek, 1341
1	MS 1179	Roger Martz, 1341
1	MS 1179	Aaron Olson, 1341
1	MS 1179	Greg D. Valdez, 1341
1	MS 1321	Daniel Ibanez, 1443
1	MS 0359	D. Chavez, LDRD Office, 1911
1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories