LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TH-821014

Enhancing Monte Carlo Particle Transport for Modern Many-Core Architectures

R. C. Bleile

March 31, 2021

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

ENHANCING MONTE CARLO PARTICLE TRANSPORT FOR MODERN MANY-CORE ARCHITECTURES

by

RYAN C. BLEILE

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

March 2021

DISSERTATION APPROVAL PAGE

Student: Ryan C. Bleile

Title: Enhancing Monte Carlo Particle Transport for Modern Many-Core Architectures

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Hank Childs	Chair
Allen Malony	Core Member
Boyana Norris	Core Member
Shabnam Akhtari	Institutional Representative

and

Kate Mondloch	Interim Vice Provost and Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded March 2021

DISSERTATION ABSTRACT

Ryan C. Bleile

Doctor of Philosophy

College of Arts and Sciences

March 2021

Title: Enhancing Monte Carlo Particle Transport for Modern Many-Core Architectures

Since near the very beginning of electronic computing, Monte Carlo particle transport has been a fundamental approach for solving computational physics problems. Due to the high computational demands and inherently parallel nature of these applications, Monte Carlo transport applications are often performed in the supercomputing environment. That said, supercomputers are changing, as parallelism has dramatically increased with each supercomputer node, including regular inclusion of many-core devices. Monte Carlo transport, like all applications that run on supercomputers, will be forced to make significant changes to their designs in order to utilize these new architectures effectively. This dissertation presents solutions for central challenges that face Monte Carlo particle transport in this changing environment, specifically in the areas of threading models, tracking algorithms, tally data collection, and heterogenous load balancing. In addition, the dissertation culminates with a study that combines all of the presented techniques in a production application at scale on Lawrence Livermore National Laboratory's RZAnsel Supercomputer.

This dissertation includes previously published co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Ryan C. Bleile

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
University of the Pacific, Stockton, CA, USA

DEGREES AWARDED:

Doctor of Philosophy, Computer and Information Science, 2021, University of Oregon
Bachelor of Science, Computer Science and Physics, 2013, University of the Pacific, Magna Cum Laude

AREAS OF SPECIAL INTEREST:

Computational Science
High Performance Computing
Nuclear Science
Scientific Visualization

PROFESSIONAL EXPERIENCE:

Software Engineer, Lawrence Livermore National Laboratory, 2018-Present
Lawrence Graduate Scholar Program, Lawrence Livermore National Laboratory, 2015-2018
Summer Student Internship, Lawrence Livermore National Laboratory, Summers 2012-2015

GRANTS, AWARDS AND HONORS:

Lawrence Graduate Scholar Fellow, Lawrence Livermore National Laboratory, 2015-2018

Lifetime member of Phi Kappa Phi honor society, 2012
Magna Cum Laude, University of the Pacific, 2013

PUBLICATIONS:

Ryan Bleile, Patrick Brantley, Matthew O'Brien, Hank Childs. "Monte Carlo Transport on GPUs at Scale." In preparation.

Ryan Bleile, Patrick Brantley, Matthew O'Brien, Hank Childs. "A Dynamic Replication Approach for Monte Carlo Photon Transport on Heterogeneous Architectures." In submission.

Matthew O'Brien, Scott McKinley, Shawn Dawson, Patrick Brantley, **Ryan Bleile**, Nick Gentile. "Hybrid CPU-GPU Load Balancing For Monte Carlo Particle Transport." The 26th International Conference on Transport Theory (ICTT-26) Sorbonne University, Paris, France. September 2019.

Bleile, R., Brantley, P., Richards, D., Dawson, S., McKinley, M. S., O'Brien, M., Childs, H. "Thin-Threads: An Approach for History-Based Monte Carlo on GPUs." In 2019 International Conference on High Performance Computing & Simulation (HPCS), pp. 273-280, Dublin, Ireland, July 2019.

M.S. McKinley, **R. Bleile**, P.S. Brantley, S. Dawson, M. O'Brien, M. Pozulp, D. Richards. "Status of LLNL Monte Carlo Transport Codes on Sierra GPUs." International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering (M&C 2019), Portland, OR. April 2019.

Richards, D. F., **Bleile, R. C.**, Brantley, P. S., Dawson, S. A., McKinley, M. S., O'Brien, M. J. "Quicksilver: A Proxy App for the Monte Carlo Transport Code Mercury." In IEEE International Conference on Cluster Computing (CLUSTER), pp. 866-873, Honolulu, HI. September 2017.

Patrick S. Brantley, **Ryan C. Bleile**, Shawn A. Dawson, Nick Gentile, M. Scott McKinley, Matthew J. O'Brien, Michael M. Pozulp, David F. Richards, David E. Stevens, Jonathan A. Walsh, Hank Childs. "LLNL Monte Carlo Transport Research Efforts for Advanced Computing Architectures." International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering (M&C 2017). February 2017.

Bleile, R., Sugiyama, L., Garth, C., Childs, H. "Accelerating advection via approximate block exterior flow maps." Electronic Imaging, pp. 140-148, Burlingame, CA. 2017

Bleile, R. C., Brantley, P. S., O'Brien, M. J., Childs, H. H. "Algorithmic Improvements for Portable Event-Based Monte Carlo Transport Using the NVIDIA Thrust Library." Transactions American Nuclear Society (ANS), vol 115, pp. 535-538, Las Vegas, NV. November 2016.

Bleile, R. C., Brantley, P. S., Dawson, S. A., O'Brien, M. J., Childs, H. "Investigation of Portable Event-Based Monte Carlo Transport Using the NVIDIA Thrust Library." Transactions American Nuclear Society (ANS), vol. 114, pp. 941-944, New Orleans, LA. June 2016.

Harrison, C., Weiler, J., **Bleile, R.**, Gaither, K., Childs, H. "A distributed-memory algorithm for connected components labeling of simulation data." Topological and Statistical Methods for Complex Data. Springer, Berlin, Heidelberg, pp. 3-19. 2015

ACKNOWLEDGEMENTS

Firstly, I would like to thank Dr. Hank Childs for the many hours of advising, writing help, and the endless dedication. Without your help I know my own writing ability would be far inferior and I will take these lessons with me for the rest of my life. Additionally, I would like to thank you for sticking with me through this process as I am the last of your first batch of students to reach this milestone.

I would like to thank the members of my dissertation committee, Dr. Allen Malony, Dr. Boyana Norris, Dr. Shabnam Ahktari, for their time, advice, and feedback ensuring the success of this dissertation.

I would like to give a special thanks to Dr. William Svoboda. You developed more than just my leadership and public speaking skills throughout high school, teaching me to build confidence in myself and pushing me to pursue my education.

I would also like to thank my undergraduate Physics professor, Dr. Sayandeb Basu, for taking a special interest in my personal development, pushing me to reach for more and strive for further education. Your endless energy and work ethic was a constant motivation for me to push myself and work hard through tough times.

I would like to thank Dr. Patrick Brantley, Dr. Matthew O'Brien, and Dr. David Richards for providing me with perspective and understanding. Your aid and many discussions about my progress enlightened me regarding the larger problems I faced pursuing this line of research. Additionally, I would like to extend my thanks to all of the Monte Carlo team providing me with access to systems, software, and support along this journey.

In addition, I would like to thank members of the CDUX research group for all of their support, and in particular the founding members: James Kress, Stephanie Labasan (Brink), Matthew Larsen, and Shaomeng (Sam) Li.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

I dedicate this dissertation to my fiancée, Alexis Fernandez, who has supported me throughout all of my years of schooling including many late nights and weekends.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

CHAPTER I

MOTIVATION

The Monte Carlo method is the name given to a class of numerical algorithms that solve problems by using pseudo-random numbers to sample probability distributions. These algorithms are frequently applied to a diverse collection of problems. Monte Carlo transport is an application of the Monte Carlo method used in conjunction with particle transport physics. In this setting, the probability distributions describe the likelihood of different particle interactions and collision reactions to occur. In order to achieve sufficient accuracy, this approach requires tracking large numbers of particles, each of which moves through a potentially large problem space. Frequently, the number of particles become so large that high performance computing (HPC) platforms are needed to complete calculations quickly enough.

Over the past decade, HPC environments have progressively moved towards many-core computing architectures. The *Top 500 List* (2019) for supercomputers is now dominated by large scale GPU or coprocessor based systems. As a result, it is now essential that applications that wish to work in the supercomputing environment also continue to adapt in order to take advantage of many-core hardware architectures. The concerns for this topic include not only taking full advantage of the compute resources, but also developing approaches that work over diverse compute hardware, including different types of GPUs. Monte Carlo transport applications are among the important applications that need to make necessary transformations in order to adapt to the change in the computing environment. However, the process to undergo this transformation is not clear. Current Monte Carlo transport applications, while parallelizable, are not well suited

to accelerator architectures: they are memory latency bound and not compute bound, they require large amounts of local memory per thread, and they have very divergent behavior.

This dissertation responds to this changing and challenging landscape. Its goal is to illuminate how to transform Monte Carlo transport algorithms to excel on many-core architectures. The primary research question and its subquestions are explained below in section 1.1 (Research Questions).

1.1 Research Questions

This dissertation answers the following primary question:

What changes to Monte Carlo particle transport algorithms will enable effective utilization of many-core architectures?

Further, this question assumes that the changes will extend the state of the art, but not remove any current capability. In particular, the changes should allow for supporting traditional HPC architectures (i.e., the code base should work on both many-core architectures and regular CPU platforms) and the resulting algorithms should work effectively for a diverse set of "problems" (i.e. workloads corresponding to different physics, geometries, and particle counts).

We approach this primary question via four subquestions, each of which answers a part of the primary question.

1. What tracking algorithms are best suited for portable performance of Monte Carlo transport on modern many-core systems?

2. What is the best way to manage data-races and the memory needs of many-core platforms?

3. Is it worthwhile to fully utilize heterogenous node architectures?

4. How does many-core focused algorithm development impact performance concerns as we scale up MPI resources?

The remainder of this section discusses further each of the research subquestions.

What tracking algorithms are best suited for portable performance of Monte Carlo transport on modern many-core systems?

This discussion is primarily focused on the open question of whether a history- or event-based tracking algorithm is better for many-core architectures and multi-core architectures. Traditionally, Monte Carlo transport algorithms have followed a history-based tracking approach. In the history-based approach a particle is assigned to a thread, which computes each event this particle will undergo for its entire lifecycle, where an event is the result of a particle moving until it must perform some interaction with the background material or mesh. This approach is easily parallelizable, as each particle is handled completely independently of the others. While this approach maps well to traditional CPU architectures, the divergence and high memory requirements make it less suited to many-core architectures. Historically, there were efforts to study event-based algorithms for the specialized vector machines of the 1990's. These algorithms took advantage of vector parallelism and the relatively high speed of memory movement compared to compute. In these methods particles were pushed onto stacks based on which event needed to be computed next. Once a stack was full or there were no more particles to sort, a vectorized calculation was done over the subgroups of particles. These methods vectorized instructions for parallelism and were relatively efficient on early vector-platforms. This approach also introduces a significant overhead related to sorting the particles after each event in order to maintain

vectorizable instructions. In the current era of computing, the cost of sorting is significantly higher as the speed of memory compared to the speed of the compute has changed significantly. This makes many of these algorithms less efficient than previously measured. In all, between historical studies and recent works, which method to use is still an open question.

What is the best way to manage data-races and the memory needs of many-core platforms?

This question can be answered by looking at data management in two parts. The first part is examining the underlying threading model that defines the way that data is managed. The second part is understanding the performance and memory tradeoffs for solutions to managing output tally memory.

The introduction of many-core architectures into the HPC landscape has significantly modified the performance characteristics of a single thread of execution. On traditional multi-core CPU architectures, individual threads have access to a significant amount of the available memory and computing power. This leads to optimizations that enable these threads to take on more work, utilize a deep cache hierarchy, and avoid the need for inter-thread communications. Schemes such as replicating data across threads are possible and provide decent performance especially compared to relying on managing access to memory via atomics, or locking mechanisms. On many-core architectures, individual threads have a lot less memory and compute capability compared to their counter parts. It is the large number of these threads that makes them powerful on the whole. In this case schemes such as the data replication are often not feasible as there is simply not enough memory to satisfy the request, nor enough compute power to effectively

utilize all the memory being requested. Due to this change we need to consider a new threading model which is designed to match these differences.

In Monte Carlo transport problems data is collected by all processors/threads in the form of tallies. This tally data exists in multiple forms: single value scalars, multi-value scalars over problem parameters such as materials or energies groups, and multi value scalars over mesh elements often also including dimensions over materials or energy groups per element. Collecting this data is the goal of a simulation and therefore needs to be performant on all architectures. There are two primary ways to deal with data management: atomic operations or replication. Managing data through atomic operations enables a single copy of memory at the expense of threads needing locked access to this memory. Replications provide threads easy access to memory at the expense of duplicating memory and necessitating reductions. To understand this space we need to weigh the costs and benefits of each approach and consider alternatives to these approaches.

Is it worthwhile to fully utilize heterogenous node architectures?

The introduction of GPU based systems, with significantly more compute capabilities on the GPUs than remains on the CPUs, has led many to consider ignoring the CPU cores on a node for computations. For applications which achieve significant performance from those available FLOPS, this is a valid path to success. Monte Carlo transport algorithms, however, are not bound by FLOPS for their performance, meaning that CPUs may be able to add performance beyond a simple consideration of FLOPS ratios. Additionally, Monte Carlo problems can often spend a lot of time on areas of the code that are not parallelizable by

GPUs, meaning that more MPI ranks are needed to achieve better performance. If applications want to consider using the CPUs and GPUs simultaneously, they must contend with the issue of how to balance the work between two architectures with different performance characteristics. Most load-balancing algorithms assume that each unit of work will perform roughly the same as any other. Thus, if particles are spread equally among all processing units the work will be roughly balanced. This is no longer an acceptable assumption to make and could be detrimental to getting performance. Understanding this question must then entail examining new algorithms to load balance between CPUs and GPUs and the effect these have on possible performance.

How does many-core focused algorithm development impact performance concerns as we scale up MPI resources?

Monte Carlo transport applications are commonly run at large scale via MPI parallelism. This process can lead to algorithms that appear to be optimized at low numbers of ranks and yet exhibit unexpected behaviors that need to be fixed in order to run at large scale. In all, because of the need to run Monte Carlo transport codes efficiently at scale, new approaches must be evaluated at scale. Only by showing the effectiveness of an approach at scale do we know for sure it is a viable approach for others to use.

1.2 Dissertation Outline

This dissertation has the following organization:

- Chapter II Background and Related Work - We discuss the relevant history and background for Monte Carlo transport followed by an in-depth survey of the computer science focused research relating to Monte Carlo transport

applications. This survey provides a baseline to understand the places where information is lacking and how we can contribute to the space.

- Chapter III Tracking Algorithms - We present our early foundational work in the discussion of event- versus history-based algorithms for Monte Carlo transport applications on GPU platforms. This work addresses the similarities and differences between GPUs and older vector architectures while also showing a new data parallel primitive method for implementing an event-based tracking algorithm.

- Chapter IV Data Race Management: Threading Models - We evaluate the concept of managing data race conditions through different threading models. In this work we present new definitions for how memory plays a role in the threading model and show results for the new concept on GPUs. Additionally, we compare the new and old threading models to ensure that the new design does not degrade performance on existing CPU platforms.

- Chapter V Data Race Management: Output Tally Data - We evaluate the impact of atomic operations on the performance of GPU kernels. In this work we present a new concept, variable replication, in which we mitigate the performance concerns of atomics with memory constraints in a GPU environment.

- Chapter VI Heterogeneous Architecture Utilization - We present a new algorithm for dynamic replication of domain decomposed problems on heterogeneous architecture. This work introduces the concept that the CPU and GPU can both provide performance and a load balancing solution to ensure that the possibility of performance is attainable.

– Chapter VII Performance at Scale - We take the best approaches from our previous results and evaluate them in a production application and at larger scale. In this section we show performance results for our production applications as it stands at the time of this writing to summarize the impact that these research elements have played in the overall success of the production application.

– Chapter VIII Conclusions and Future Work - We conclude by summarizing our contributions and their overall impact on both our application and the Monte Carlo community as a whole. In addition, we present a discussion of future work which could extend the works presented in this dissertation.

Co-Authored Material

Much of the work in this dissertation is from previously published and unpublished co-authored material. Below is a listing connecting the chapters with the publications and authors that contributed. For each of these publications, I was not only the first-author of the paper, but also the primary contributor for implementing software, conducting studies, and writing manuscripts.

– Chapter I: This chapter is composed of portions of my dissertation proposal, which was unpublished. Dr. Hank Childs participated in discussing the material and editing.

– Chapter II: This chapter is composed of portions of my Ph.D. Area Exam, which was unpublished. Dr. Hank Childs participated in discussing the material and editing.

– Chapter III: This chapter is a combination of two ANS short papers, Bleile, Brantley, Dawson, O'Brien, and Childs (2016) and Bleile, Brantley,

O'Brien, and Childs (2016). I was the primary contributor to this work in developing the algorithm, writing the new code, and writing the paper. Dr. Patrick Brantley initially identified the need for this work and provided the application that this work was performed in. Dr. Patrick Brantley, Dr. Shawn Dawson, and Dr. Matthew O'Brien provided ideas and feedback throughout the development process and assisted in editing the paper. Dr. Hank Childs assisted in editing the paper.

- Chapter IV: This chapter is primarily from work published at HPCS, Bleile et al. (2019). I was the primary contributor to this work in developing the algorithm, writing the new code, and writing the paper. Dr. Hank Childs, Dr. Patrick Brantley and Dr. Matthew O'Brien provided ideas and feedback throughout the development process and assisted in editing the paper.

- Chapter V: This chapter is from a study done for a paper that never was published. I was the only contributing author on this effort. Dr. David Richards provided insight and discussion during development.

- Chapter VI: This chapter is from a work which is in submission to ICCS, Bleile, Brantley, O'Brien, and Childs ((in-submission) 2021). I was the primary contributor to this work in developing the algorithm, writing the new code, and writing the paper. Dr. Hank Childs, Dr. Patrick Brantley and Dr. Matthew O'Brien provided ideas and feedback throughout the development process and assisted in editing the paper.

- Chapter VII: This chapter is from a work which is in progress to be submitted, Bleile, Brantley, O'Brien, and Childs (2021). I was the primary contributor to this work in designing and running the tests as well as writing

the content of this chapter. Dr. Hank Childs, Dr. Patrick Brantley and Dr. Matthew O'Brien provided ideas and feedback throughout the development process and assisted in editing the written content.

CHAPTER II

BACKGROUND AND RELATED WORK

2.1 Introduction

This chapter considers Monte Carlo particle transport with respect to modern supercomputers. While Monte Carlo particle transport is well understood, today's supercomputer landscape is in flux. Supercomputer architectures are undergoing more extreme changes now than at any point in the past twenty years. An important driving factor for this change is the concern regarding power usage while scaling to larger and larger machines. Modern machines are pushing up against a hard power limit, meaning that in order to increase performance they must become more power efficient. As a result, architectures are transitioning from fast and complex multi-core CPUs to the more energy efficient design of larger numbers of slower and simpler processors. Relative to one decade ago, the amount of parallelism available on any given node in a supercomputer is growing by factors of hundreds or thousands because of this change. This transition to many-core computing brings new and interesting challenges

This chapter is organized into four parts: the first part provides a background in Monte Carlo particle transport, the second part continues with a discussion of the current state of the art research for Monte Carlo particle transport calculations, the third part focuses exclusively on Monte Carlo on GPUs, and the fourth part considers portable performance across many architectures.

2.2 What is Monte Carlo Particle Transport?

Eckhardt (1987) provides the unpublished conversation Stan Ulam and John von Neumann had discussing a game of solitaire which became the foundation for starting Monte Carlo transport methods.

"The first thoughts and attempts I made to practice [the Monte Carlo method] were suggested by a question which occurred to me in 1946 as I was convalescing from an illness and playing solitaire. The question was what are the chances that a Canfield solitaire laid out with 52 cards will come out successfully? After spending a lot of time trying to estimate them by pure combinatorial calculations, I wondered whether a more practical method than "abstract thinking" might not be to lay it out say one hundred times and simply observe and count the number of successful plays. This was already possible to envisage with the beginning of the new era of fast computers, and I immediately thought of problems of neutron diffusion and other questions of mathematical physics, and more generally how to change processes described by certain differential equations into an equivalent form interpretable as a succession of random operations. Later... [in 1946, I] described the idea to John von Neumann and we began to plan actual calculations."

- Stan Ulam 1983

John von Neumann became interested in Stan Ulam's idea and outlined how to solve the neutron diffusion and multiplication problems in fission devices. Since this time, Eckhardt (1987) tells us that Monte Carlo methods have continued to be a primary way for solving many questions in neutron transport.

2.2.1 Definition. In Computational Methods of Neutron Transport, Lewis and Miller (1993) describe Monte Carlo transport as a simulation of some number of particle histories by using a random number generator. For each particle history that is calculated, random numbers are generated and used to sample probability distributions describing the different physical events a particle can

undergo, such as scattering angles or the length between collisions. Lux and Koblinger (1991) further expand the previous definition in their book Monte Carlo Particle Transport Methods: Neutron and Photon Calculations:

> "In all applications of the Monte Carlo method a stochastic model is constructed in which the expected value of a certain random variable (or of a combination of several variables) is equivalent to the value of a physical quantity to be determined. This expectation value is then estimated by the averaging of several independent samples representing the random variable introduced above. For the construction of the series of independent samples, random numbers following the distributions of the variable to be estimated are used." (p. 5)

Lewis and Miller (1993) explain that estimating a quantity takes on the following mathematical form:

$$\hat{x} = \frac{1}{N} \sum_{n=1}^{N} x_n,$$

where x_n represents the contribution of the nth history for that quantity. For the Monte Carlo method, we tally the x_n from each particle history in order to compute the expected value $\hat{x}$.

One very important question is how the estimated value $\hat{x}$ compares to the true value $\bar{x}$. It turns out that the uncertainty in $\hat{x}$ decreases with increasing numbers of particle histories, and generally falls off asymptotically proportionate to $N^{-1/2}$, where N is the number of particles.

2.2.2 The Equation. Monte Carlo neutron transport solves the equation known as the Linearized Boltzmann transport equation. Large numbers of particles are used to create accurate estimations for each of the quantities that

make up this equation. Since the Boltzmann equation can be written down in different ways we focus on the variation described by Lewis and Miller (1993). The equation and components descriptions are:

$$\left[\frac{1}{\nu}\frac{\partial}{\partial t} + \hat{\Omega}\cdot\vec{\nabla} + \Sigma_t(\vec{r},E)\right]\Psi(\vec{r},\hat{\Omega},E,t) =$$

$$S_{ext}(\vec{r},\hat{\Omega},E,t) \; + \; \int_{E'}\int_{\Omega'} \Sigma_s(\vec{r},E'\rightarrow E,\hat{\Omega}'\cdot\hat{\Omega})\Psi(\vec{r},\hat{\Omega}',E',t)d\Omega' dE' \; +$$

$$\chi(E)\int_{E'}\nu(E')\Sigma_f(\vec{r},E')\int_{\Omega'}\Psi(\vec{r},\hat{\Omega}',E',t)d\Omega' dE'$$

where $\Psi(\vec{r},\hat{\Omega},E,t)$ is the angular flux, $\Sigma_t(\vec{r},E)$ is the macroscopic total cross section for all particle reactions, $\Sigma_s(\vec{r},E' \rightarrow E,\hat{\Omega}'\cdot\hat{\Omega})$ is the macroscopic cross section for particle scattering, $\Sigma_f(\vec{r},E)$ is the macroscopic cross section for particle production from a fission reaction, $\chi(E)$ is a secondary particle spectrum from the fission process, $\nu(E)$ is the average number of particles emitted per fission, $S_{ext}(\vec{r},\hat{\Omega},E,t)$ represents an external source, $\vec{r}$ is the spatial coordinates, E is the energy, $\hat{\Omega}$ is angular direction, and t is time.

2.2.3 Algorithmic Approach. There are many ways to solve this problem. The most common method is to track individual particle histories until a predetermined amount of particles have been simulated. This method is known as the history-based approach. In order to simulate a particle, the distance the particle must travel before it has any interactions must be computed and compared with all possible interactions. The interaction with the shortest distance is chosen, followed by updating the particle and tallies based on the distance traveled and interaction occurring. Algorithm 1 shows the history-based approach for a simple research code as defined by Bleile, Brantley, Dawson, et al. (2016). Algorithm 2 shows the outer most scope of a Monte Carlo problem, both for providing context

on Algorithm 1's placement and for emphasizing where different optimizations can occur. In particular, Algorithm 1 takes place inside the Cycle loop of Algorithm 2 and shows only the steps for Cycle: Tracking.

Algorithm 1: History-based Monte Carlo tracking algorithm

```
1  foreach particle history do
2      while particle not escaped or absorbed do
3          sample distance to collision in material
4          sample distance to material interface
5          compute distance to cell boundary
6          select minimum distance, move particle, and perform event
7          if particle escaped spatial domain then
8              update leakage tally
9              end particle history
10         if particle absorbed then
11             update absorption tally
12             end particle history
```

Algorithm 2: Monte Carlo method

```
1  parse inputs
2  foreach Cycle do
3      initialize
4      tracking (Algorithm 1)
5      finalize
6  gather tallies
```

While the history-based algorithm is the most common approach, it is not the only possible approach to tracking particles. Event-based variations to this algorithm exist and are an open point of research, both from the 1980's and 1990's and again in the context of modern GPU machines. In an event-based approach particles are grouped together and each operations is applied to all or a subset of them at once. Some variations of this algorithm are discussed in subsection 2.4.4 (Event-Based Techniques) and then again in Chapter III (Tracking Algorithms).

2.3 State of the Art: Monte Carlo Research

There is a long history of research and improvements for Monte Carlo transport problems. Further, understanding this path, and the machines that the approaches were designed for, helps to guide analysis of more recent efforts. Most recent research efforts for Monte Carlo transport have been related to one of these three topics: (1) GPGPU computing, (2) parallel algorithm improvements (not concerning GPGPUs), or (3) physics improvements. A review of the GPGPU related research is presented in section 2.4 (State of the Art: GPU Research). This section focuses on the non-GPU Monte Carlo transport research, namely parallel performance on CPU architectures, parallel load balancing, optimizations in nuclear data look-ups, and variance reduction techniques.

2.3.1 Parallel Performance. Since the inception of Monte Carlo particle transport over sixty years ago, there has been tremendous growth as Monte Carlo applications have adapted to maximize performance on each new generation of architectures. The first models developed in 1947 would take five hours to compute 100 collisions, a task that today can be done in milliseconds. In the 1940's and 1950's, Monte Carlo codes were written in very low level languages on the earliest computers. The 1960's to 1980's saw a great increase in the capabilities of the Monte Carlo codes as computing power increased and codes become more fully featured. In the 1980's Monte Carlo codes adopted parallel/vector machines. In the 1990's Monte Carlo codes become more commonplace and parallelism increased to 100s or 1000s or processors through message passing with *PVM Parallel Virtual Machines* (2011) or the Message Passing Interface (Clarke, Glendinning, and Hempel (1994)). In the 2000's, Brown (2011) explains that multicore processors

meant threading became more commonplace, mixing local and global forms of parallelism reaching tens of thousands of processors.

This growth in computer processing can also be categorized in terms of the styles of memory accesses. Early systems were shared-memory environments almost exclusively. Then distributed-memory systems became popular. Finally, hybrid parallel systems, meaning systems that use both distributed- and shared-memory parallelism, became popular. The following discussion is organized around these three types of parallelism.

Shared-Memory Performance. Shared-memory systems refer to machines or models where all processors can access the same memory space. Taking this a step further, in the unified memory architecture (UMA), El-Rewini and Abd-El-Barr (2005) explain that all processors have access to the same memory and access to all memory takes the same amount of time. One type of shared-memory system that was popular was the vector machine. Russell (1978) explains that vector machines took the shared-memory system and added additional synchronicity to the system by making all of the processors issue the same instruction. This type of parallelism is often referred to as SIMD or single instruction multiple data, meaning that the same instruction is going to affect multiple data elements at once.

Vector Machine Performance. The research discussion presented in this subsection occurred many years ago, but maintains relevance as modern architectures trend towards similarities with architectures popular during this time. GPUs, while not strictly SIMD, operate in a comparable fashion with vector machines as they force instructions to operate in a lock step fashion. Additionally, Intel MIC architectures are vector processors, with the primary difference being

a significant update in how memory is managed on these devices versus previous developments. Due to this similarity, a discussion of research from the vector machine era is merited.

In the 1980's Monte Carlo transport algorithms began adapting event-based methods. The traditional history-based approach was not well suited for vector architectures, since the particle histories follow independent code paths. In order to vectorize the algorithms and make them usable on the vector machine architectures, the codes had to be reorganized to follow the same code paths across independently computed particle histories. Martin, Nowak, and Rathkopf (1986) explain that by changing the algorithm to follow events instead of histories, the Monte Carlo method could be used in a vector based approach.

A common element for work in this area is that the vector approach is often related to a stack data structure. The challenge then becomes properly organizing particles into the right sub-stack so that calculations can be performed as explained in works by Brown and Martin (1984) and also by Bobrowicz, Lynch, Fisher, and Tabor (1984). Martin et al. (1986) then tried another approach in which there is only one main particle stack and only the minimum information needed to compute each of the events is pulled off into sub-stacks. With each of these approaches, particle events determine not only how the particles are organized but also what information is needed for processing. The main drawback to the event-based approach is the added time for data movement or sorting.

Brown and Martin (1984) reported the potential for speedups of 20X-85X in their theoretical analysis of the use of event-based methods, and deemed this approach well worth the efforts required to refactor codes in order to use this approach on vector machines. Martin et al. (1986) saw speedups ranging from 5X

to 12X using the single big stack, sub-stack approach, depending on the problem choice and the machine he was running on. Vujic and Martin (1991) vectorized and parallelized a reactor assembly code using explicit stacks and vectorizable data structures, reducing wall-clock time by 22X. Bobrowicz et al. (1984) implemented an explicit stack approach and reached speedups of around 8X - 10X compared with the original history-based approach. Finally, Burns, Christon, Schweitzer, Lubeck, and Wasserman (1989) used GAMTEB, the LANL Benchmark code, to demonstrate similar performance to Bobrowicz et al. (1984) by following an approach similar to Brown and Martin (1984).

Multi-Threaded Architecture Performance. Other shared-memory systems, separate from vector machines, were tried in this era. One such machine was the Tera Multi-Threaded Architecture (MTA) as explained by Snavely et al. (1998). This approach focused on the use of parallel processors, hardware threading, and a shared-memory no cache design. The idea was to mask away memory latency by focusing on threading. This concept is shared in modern GPU systems. As memory latency is high among individual threads, latency hiding through massive parallelism is a focus of these devices. Due to this similarity it is worth understanding the work done in this space.

Majumdar (2000) tried two methods of parallelizing the photon transport application TPHOT on the Tera MTA. In TPHOT, looping occurred over spatial zones and all possible energy levels, with photons being computed when their containing zone/energy level was processed. So for this application Majumdar chose to parallelize both over zones and over a combination of zones and energies through loop unrolling. Table 1 shows that the parallelization on the MTA over zones and energies maintains incredible efficiency giving their application good

Table 1. Parallel performance of TPHOT on the Tera MTA using multithreading by Majumdar (2000)

Procs	Time (sec)	Speedup	Efficiency
Parallelization by zones only			
1	764	1.00	1.00
2	400	1.91	0.95
4	227	3.37	0.84
8	167	4.58	0.57
Parallelization by zones and energies			
1	745	1.00	1.00
2	370	2.01	1.01
4	187	3.98	0.99
8	94	7.92	0.99

speedups, while parallelizing over only zones does not expose enough parallel work to hide memory latency and so efficiency drops off quickly.

More modern systems utilize shared-memory ideas as well, with a majority of the scientific efforts utilizing OpenMP threading models for shared-memory processing. Often this model is overlooked in preference of distributed computing via MPI but that is not always the case. Given an all particle method, OpenMP codes tend to scale with good efficiency with the only drawbacks having to do with possible atomic operations occurring during the collection of output tallies. In the case of no atomic operations and plenty of work, Siegel, Smith, Romano, Forget, and Felker (2014) showed that shared-memory Monte Carlo transport implementations have nearly perfect efficiency on a node.

Distributed-Memory Performance. One of the major transitions in supercomputing history came with the shift from vector computing to distributed-memory computing. This type of computing is most often done with MPI and has, for the last 20 years, been a primary method of achieving parallel performance on

small scale compute clusters and large scale supercomputers alike. In the message passing model of parallelism, independent processes work together through the use of messages to synchronize actions or pass data between processors. Yang, Yu, and Wang (2015) explain that, in this model, parallel efficiency is generally improved by spending more time working independently and is negatively affected by time spent sending messages or idled at a synchronization point.

Yang et al. (2015) tell us that the Monte Carlo particle transport history-based approach lends itself to the distributed model very well. Each particle history is independent of any other particle histories and can be easily split up over processors. However, moving to distributed memory systems creates complications in handling large and complex geometries. Domain decomposition is typically used in this case, although it increases the complexity in using message passing. Domain decomposition challenges are discussed further in subsection 2.3.2 (Load Balance and Domain Decomposition).

Given the embarrassingly parallel nature of the Monte Carlo transport problem, the performance of this model produces results as expected. As MPI processes increase, Monte Carlo transport continues to get a nearly linear speedup. Majumdar (2000) shows that with 16 nodes and 8 MPI tasks per node, his biggest run, he was still able to achieve a 88% efficiency in his code that was parallel over zones and energies. M. J. O'Brien, Brantley, and Joy (2013) demonstrate that in an all particle code, Mercury at LLNL, parallel efficiencies of $\sim$95% are seen when using MPI parallelism up to 2 million processors.

Distributed- + Shared-Memory Performance. Given the heterogenous nature of today's computing environment, and even in the fairly homogenous environment that has been prevalent, it is a common next step to

consider combining distributed and shared-memory parallel schemes. Shared-memory parallelism exists not only on the CPU cores of a node, but also on the threads of a many-core accelerator. Distributed-memory parallelism provides the opportunity for scaling to large supercomputers or clusters, giving users many nodes to work with. Given the natural fit between these two models it is surprising how rarely they are combined in practice. Developers may have avoided using both types of parallelism to date since distributed-memory approaches work "well enough" within a node, meaning that each core on a node can be its own MPI task. With the addition of accelerator architectures developers will no longer be able to ignore combining shared-memory and distributed-memory models

Wolfe (2014) defines the combined distributed+shared model as MPI+X. The X in this description being replaced with whichever shared-memory system is preferred. The most common implementation of MPI+X to date is the MPI + OpenMP model. Wolfe (2014) explains that, in the MPI + OpenMP model, MPI is utilized for node to node communication and OpenMP is used for on node parallelism.

Yang et al. (2015) has recently shown that the MPI+OpenMP model has the benefit of achieving nearly perfect parallel efficiency and of significantly decreasing the memory overhead to an equivalent MPI only implementation. He was able to show 82-84% parallel efficiency and a decrease in memory cost from $\sim$1.4GB to $\sim$200MB for 8 processors. While Majumdar (2000) shows that with 16 nodes and 8 OpenMP threads per node, he was able to achieve a 95% parallel efficiency which is an improvement over his MPI only method's 88% parallel efficiency.

2.3.2 Load Balance and Domain Decomposition.

In order to achieve high levels of parallelism in transport problems with many geometries or zones, different parallel execution models are used. The two primary models used are domain decomposition and replication. Domains are sections of the problem space or geometry used for organizing where portions of a larger data set are stored. Procassini, O'Brien, and Taylor (2005) define domain decomposition as the spatial decomposition of the geometry into domains, and then the assigning of processors to work on specific domains. Additionally, Procassini et al. (2005) define replication as storing the geometry information redundantly on each processor and assigning each processor a different set of particles.

Load balance of domain replication problems is often simply a trivial splitting of particles across processors. Because of this, load balance is often discussed in conjunction with domain decomposition specifically. Particles often migrate between different regions of a problem, meaning not all spatial domains will require the same amount of computational work. In many applications there is at least one portion of the calculation that must be completed by all processors before all the processors can move forward with the calculation. As a result, if one processor has more work than any other, all of the others must wait for that processor to complete its work. M. J. O'Brien et al. (2013) demonstrated that this load imbalance can cause significant issues with scalability as parallelism is increased from hundreds to millions of processors.

When to Load Balance. A key consideration for performing a load balanced calculation is understanding the cost of performing that calculation. If too much time is spent making sure the problem is always perfectly load balanced, then computational resources are being wasted on a non-essential calculation,

resulting in overall slower performance. However, if too few resources are devoted to load balancing then the problem will suffer from load imbalance and the negative effects that entail. M. O'Brien, Taylor, and Procassini (2005) provide one solution, which is to perform load balance at the start of each cycle or iteration of a Monte Carlo transport calculation, but only when that load balance will result in a faster overall calculation. Their method to determine when to load balance is to use a criterion that can be checked inexpensively each cycle to determine if a load-balance operation should take place. The first step is to compute a speedup factor by comparing current parallel efficiency (ε_C) to what parallel efficiency would be if processors were to redistribute their load (ε_{LB}). The second step is to predict the run time by using the time to execute the previous cycle (τ_{Phys}), the speedup factor (S), and finally, the time to compute the load balance itself (τ_{LB}). The final step is to compare the predicted runtime with and without load balancing to determine if the operation is worthwhile. The equations describing this algorithm then are:

$$S = \frac{\varepsilon_C}{\varepsilon_{LB}} \tag{2.1}$$

$$\tau^{'} = \tau_{Phys} \cdot S + \tau_{LB} \tag{2.2}$$

$$\tau = \tau_{Phys} \tag{2.3}$$

$$if\ (\tau^{'} < 0.9 \cdot \tau\ )\ DynamicLoadBalance() \tag{2.4}$$

Extended Domain Decomposition. As an extension to the domain decomposition of meshes, M. O'Brien, Joy, Procassini, and Greenman (2009) demonstrated an algorithm to domain decompose Constructive Solid Geometry (CSG) in a Monte Carlo transport code. One key difference between mesh and CSG geometries is that mesh geometries contain a description of cell connectivity,

whereas cells defined through CSG do not. In order to domain decompose these CSG cells, each cell was given a bounding box; using this bounding box, a test for if a cell belongs inside a domain becomes an axis-aligned box-box intersection test.

Greenman, O'Brien, Procassini, and Joy (2009) demonstrated that, in addition to pure mesh and pure CSG problems, other combinations are sometimes beneficial, such as the combination of mesh and CSG problems where there are large-scale heterogeneous and homogeneous regions. In this method, a mesh region is embedded inside a CSG region allowing for the use of either, based on whichever region is more optimal.

Load Balance at Scale. When load balancing massively parallel computers, examining the workload of every processor can affect scalability. M. J. O'Brien et al. (2013) present a scalable load balancing algorithm that runs in $\Theta(log(N))$ by using iterative processor-pair-wise balancing steps that ultimately lead to a balanced workload. Their algorithm demonstrated scalability on up to two million processors on the Sequoia supercomputer at Lawrence Livermore National Laboratory.

M. J. O'Brien et al. (2013), using the pair-wise load balancing scheme, maintained efficiency of 95% at 2 million processors, while the runs without load balancing dropped in efficiency to around 68% at 2 million processors. In addition, the load-balanced version was able to maintain near perfect scaling up to 2 million processors. By dispersing the workload effectively over processors it also decreased the overall tracking time.

Algorithms that interact with particles and geometries are affected when domain decomposition is added. M. O'Brien and Brantley (2015) describe three algorithms which are modified when domain decomposition is added. Specifically a

Global Particle Find algorithm, a Test For Done algorithm, and Domain Neighbor Replication. The Global Particle Find algorithm is used to find where a particle is currently located in the geometry. After domain decomposition a tree search was added to quickly decide which domain a particle is in before then searching in specific geometry elements. The Test For Done algorithm, which reports if there are any particles left to process, can be easily achieved by using MPI_I allreduce() in place of a complex hand coded algorithm. Lastly, the Domain Neighbor Replication was found to be effective when combined with domain decomposition, as it increased achieved load balance and reduced the total memory usage.

2.3.3 Nuclear Data. Nuclear data provides simulations with information for how materials respond to interactions with particles under a variety of conditions. It consists of microscopic cross section data for nuclear and atomic collisions for all possible reactions. Additionally, macroscopic cross section information can be calculated from the microscopic cross sections data. Both microscopic and macroscopic cross section information is needed in order to understand what reactions a particle undergoing a collision will do.

Looking up nuclear data information is a large part of Monte Carlo transport calculations, both in terms of execution time and heavy usage through many stages of the code. Nuclear data is stored in large tables of information that are generally interpreted and processed in specialized libraries. McKinley and Beck (2015) give an example of one of these libraries, the GIDI library at LLNL, which responds to all of the nuclear data requests from simulation codes. Tramm, Siegel, Islam, and Schulz (2014) show that, depending on the specific problem being solved, the time spent looking up nuclear data can vary greatly, ranging from 10% to 85% of the overall runtime. Romano and Forget (2013) and Romano et al.

(2015) provide additional examples of this behavior in the Monte Carlo application OpenMC.

There are two primary methods for storing and looking up nuclear cross section data: continuous energy and multi-group. The continuous energy model spends more time looking up cross section data since energy values are stored as a large sequence of points and exact values are found through interpolation. Multi-group cross section data is stored in some number of bins and all energies that land in the bin are given the same value. This method is often much faster, sometimes reducing searches by orders of magnitude, but less accurate.

Research that deals with nuclear data lookups is often concerned with speeding up the search for a given cross section at a given energy. This search problem is the main bottleneck in cross section lookup algorithms. Linear searches, binary searches, and hash-based searches are often employed. In addition, combining isotopes into a unionized grid is a common method for reducing the total number of searches required, though it greatly increases the memory needed to store the cross section data. Wang, Brun, Malvagi, and Calvin (2016) defined and compared the following algorithms for continuous energy model access:

Hashing. Each material's whole energy range is divided up into N equal intervals, and for every individual isotope inside the material an extra table is established to store isotopic bounding indexes of each interval. The new search intervals are thus largely narrowed with respect to the original range and can be reached by a single float division. The hashing can be performed on a linear or logarithmic scale; the search inside each interval can be performed by a binary search or linear search. In the original paper by Brown (2014), a logarithmic

hashing was chosen with $N \simeq 8000$ as the best compromise between performance and memory usage. Another variant is to perform the hashing at the isotope level.

Unionized grid. A global unionized table gathers all possible energy points in the simulation and a second table provides their corresponding indices in each isotope energy grid as defined by Leppänen (2009). Every time an energy lookup is performed, only one search is required in the unionized grid and the isotope index is directly provided by the secondary index table. Lund and Siegel (2015) provide timing results which show that this method has a significant speedup over the conventional binary search but can require up to a 36× more memory space.

Fractional cascading. This is a technique to speedup search operations for the same value in a series of related data sets. Lund and Siegel (2015) explains that the basic idea is to build a unified grid for the first and second isotopes, then for second and third isotopes, etc. When using the mapping technique, once the energy index in the first energy grid is found all the following indices can be read directly from the extra index tables without further computations. Compared to the global unionized methods, the fractional cascading technique greatly reduces memory usage.

2.3.4 Variance Reduction Techniques. Variance reduction is a key concept in Monte Carlo transport problems. The solutions to Monte Carlo problems are given in the form of statistics and so reducing the variance in those statistics leads to more accurate or easier to compute solutions. Often without some use of variance reduction, certain problems would take an incredible amount of time and computing power to find a solution. Kahn and Marshall (1953) explain that the idea behind variance reduction is to increase the efficiency of Monte Carlo

calculations and permit the reduction of the sample size in order to achieve a fixed level of accuracy or increase accuracy at a fixed sample size. Some commonly used variance reduction techniques are common random numbers, antithetic variates, control variates, importance sampling and stratified sampling, although most common methods used in Monte Carlo transport are some variation of importance sampling.

Common Random Numbers. Kahn and Marshall (1953) define this method of variance reduction, which involves comparing two or more alternative configurations instead of only a single configuration, also referred to as correlation of samples. Variance reduction is achieved by introducing an element of a positive correlation between the sets. This can be accomplished through ensuring that all configurations of a problem use the same random numbers to find solutions. The article *Variance reduction* (2021) provides a clear example: "in queueing theory, if we are comparing two different configurations of tellers in a bank, we would want the (random) time of arrival of the Nth customer to be generated using the same draw from a random number stream for both configurations."

Antithetic Variates. *Antithetic Variates* (2021) is a method of variance reduction which involves taking the antithetic path for each path sampled — so for a given path $\{\varepsilon_1, ..., \varepsilon_M\}$ one would also take the path $\{-\varepsilon_1, ..., -\varepsilon_M\}$. This method reduces the number of samples needed and reduces the variance of the sampled paths.

Control Variates. *Control Variates* (2021) is a method of variance reduction which involves creating a correlation coefficient by using information about a known quantity to reduce the error in an unknown quantity. This method

is equivalent to solving a least squares system and so is often called regression sampling.

Importance Sampling. Kahn and Marshall (1953) defines this method of variance reduction which involves estimating properties of a particular distribution, while only having samples generated from a different distribution than the distribution of interest. This method emphasizes important values by sampling them more frequently and sampling unimportant values less frequently. Melnik-Melnikov and Dekhtyaruk (2000) explains that this is often achieved through methods known as splitting or Russian roulette. In splitting and Russian roulette particles are each given a weight and if particles enter an area of higher importance they are split into more particles with less weight giving a larger sample size. If particles travel in a region that is not important they undergo Russian roulette where some particles are killed off and others are given more weight to account for those removed.

Stratified Sampling. *Stratified Sampling* (2021) is a method of variance reduction which is accomplished by separating members of a population into homogeneous groups before sampling. Sampling each stratum reduces sampling error and can produce weighted means that have less variability than the arithmetic mean of a simple sampling of the population.

Recent research in the area of variance reduction techniques often includes a specific problem that requires a more focused study to utilize one of these previously described patterns. For example, in the problem of atmospheric radiative transfer modeling, Iwabuchi (2015) recently published work describing a proposal for some variance reduction techniques that they can use to help solve the problem of solar radiance calculations. Described are four methods that are developed

directly from their problem. The first is to use a type of Russian roulette on values that will contribute small or meaningless amounts to the overall calculation within some threshold. Other methods include approximation methods for sharply peaked regions of the phase space, forcing collisions in under sampled regions, and numerical diffusion to smooth out noise.

2.4 State of the Art: GPU Research

This section considers recent advances in Monte Carlo research on GPU architectures. It first surveys different approaches for utilizing the GPU. It then surveys Monte Carlo transport from the medical transport perspective in order compare approaches from the different communities. The section goes on to survey uses of ray tracing within a Monte Carlo transport application. Finally, this section will survey new algorithm choices through event-based Monte Carlo transport.

Comparing CPU and GPU Architectures.

> "A simple way to understand the difference between a CPU and GPU is to compare how they process tasks. A CPU consists of a few cores optimized for sequential serial processing while a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously." – *What is GPU Computing?* (2015)

A CPU has been developed to optimize the performance of a single task. In order to accomplish this CPUs have been latency optimized, meaning that the time to complete one task, including gathering the necessary memory, has been reduced in any way possible. GPUs, on the other hand have been throughput optimized in order to complete as many tasks as possible in a given amount of time. This means that the time for a GPU to complete a single task is most likely significantly longer

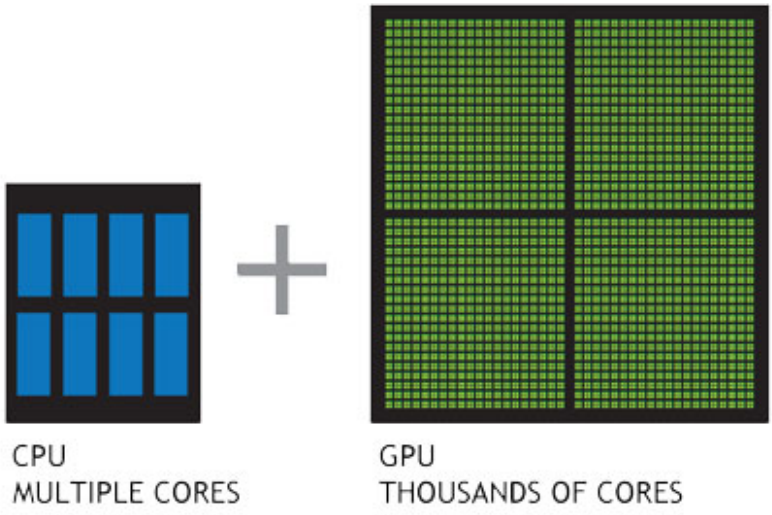


Figure 1. A visual representation of the difference between CPU and GPU hardware

than for a CPU, but, in a fixed amount of time the GPU will be able to accomplish many more tasks. So given a large enough number of tasks that can be carried out in parallel, the GPU can likely execute faster.

2.4.1 Monte Carlo Transport on a GPU. This subsection analyzes the different initial approaches Monte Carlo transport codes have taken in order to utilize GPU architectures. It begins by comparing and contrasting different approaches by evaluating a few key areas of the studies that have been done: accuracy, performance and algorithmic choices. Following is an evaluation of the effectiveness of the approaches for the range of problems being addressed. As a side note it is important to notice that speedups reported come from each paper on the hardware they were using at the time of their study; GPU hardware has changed in computing power dramatically over the last ten years in terms of performance and additional features.

Accuracy. One of the first considerations the scientific community has when being introduced to a new computing platform is what levels of accuracy can they achieve with their simulation codes. Since the change from CPU to GPU computing brings a completely different hardware design it is important to understand how that design might affect the accuracy of any calculations it is performing. This concern was especially important in the early days of GPU computing when double-precision was not supported and often even single-precision answers would provide slightly different results. There are three key areas of accuracy to consider: Floating point precision, differences between CPU and GPU results, and IEEE-754 compliance.

It was commonly assumed in the early stages of GPU computing that accuracy was lacking. Many early attempts at GPU computing included discussions of accuracy in order to validate the correctness of their results. While modern GPGPUs support double-precision much better than before, making much of the worry irrelevant, it is still important to consider the accuracy of a method that runs on a new hardware and may use a new algorithm.

Floating Point Accuracy. One of the primary concerns of the early GPU studies involved understanding the limits of floating point arithmetic on the GPU architecture. Nelson (2009) describes one of his primary accuracy considerations as being the difference between single and double-precision calculations. In older GPU hardware there was no support for double-precision in the hardware and so in order to achieve double-precision significantly more calculations were needed. In modern GPU hardware 64 bit double-precision is becoming increasingly better supported and in the GPGPU cards there are dedicated double-precision units.

Differences Between CPU and GPU results. An even larger concern than the differences between single and double-precision is differences in results that arise when using the same precision but on different architectures. Goldberg (1991) explains that this concern can be understood by considering how floating-point math is accomplished on a computer. There are two main reasons that differences arise. The first is that floating point mathematical operations performed in different orders might produce different results and, due to the nature of parallel computing, the ordering of these calculations is not guaranteed. The second reason is that modern day CPUs using x86 processors perform math internally on 80 bit registers while a GPU does it on 32 bit (single-precision) or 64 bit (double-precision) registers. Because of this, each math operation on a CPU might stay in registers and only be rounded down to 64 bits when it is saved to memory.

Jia et al. (2010) showed that in their development of a Monte Carlo dose calculation code they could achieve speedups of 5 to 6.6 times over their CPU version while maintaining within 1% of the dosing for more than 98% of the calculation points. They considered this adequate accuracy to consider using GPUs for doing these computations. Yepes, Mirkovic, and Taddei (2010) also considered accuracy in their assessment of their GPU implementation. They concluded that, in terms of accuracy, there was a good agreement between the dose distributions calculated with each version they ran, with the largest discrepancies being only $\sim$3%, and so they could run the GPU version as accurately as any general-purpose Monte Carlo program. As these two groups have shown, this amount of error is often very small, and over the entire course of the simulation only brings 1-3% errors.

IEEE-754 Compliance. Nelson (2009) discussed accuracy in his thesis work as floating-point arithmetic accuracy was not fully IEEE-754 compliant during the time of his work. Additionally, since NVIDIA has complete control over the implementation of floating point calculations on their GPUs there may be differences between generations that mitigate the usefulness of an accuracy study on one generation of hardware. Current generations of the NVIDIA GPU hardware are IEEE-754 compliant however. In order to address issues of floating point accuracy NVIDIA included a detailed description of the standard and their implementation in CUDA. This description as well as details about how NVIDIA chose to follow the standard can be found in the CUDA Toolkit, in the section *Floating Point and IEEE 754* (2015). So, while floating point accuracy is still a concern, it is now no more a concern than it was on a CPU implementation.

Performance. Performance is a second important factor for Monte Carlo transport on GPUs. Most early GPU studies emphasize their speedups over CPUs as the primary advantage for moving over to the GPU hardware. Given the change in supercomputing designs these comparisons have become increasingly more important.

Often, performance is compared to the hardware maximums such as peak of FLOPS or memory bandwidth. It is often assumed that an increase in available FLOPS will translate directly into incredible performance gains. V. W. Lee et al. (2010) in his article "Debunking the 100X GPU vs. CPU myth", brings this discussion of performance into new light, showing the relative performance gains for different types of applications. The important thing to consider is the limiting factor between the hardware and the code. As a result, comparing current performance with that of peak performance is often very misleading.

The following discussions show the relative performances of Monte Carlo transport applications that either underwent a transformation to use GPUs or performed a study comparing with GPU hardware. We will not see the 100x performance that is often sought after, but instead we can understand the impact that each applications problem, algorithms, and implementation differences had on the performance as a whole.

Photon Transport. Badal and Badano (2009) present work on photon transport in a voxelized geometry showing results around 27X over a single core CPU. Their work emphasizes the use of CUDA for GPU performance on radiograph problems.

Ren et al. (2010) present work on photon propagation through tissue, showing around a 10X performance increase when using the GPU. Their discussion expressed clearly that the performance was related strongly to the size of the data set and the number of simulated photons. In addition, their results were negatively affected by high level divergence when processing different types of tissues.

Alerstam, Svensson, and Andersson-Engels (2008) presented work on a GPU based photon migration simulation in CUDA with speedups around 1000X over a single core CPU. This specific problem does not suffer from the same divergence issues that other Monte Carlo codes have as the algorithm for completing a photon migration has very little divergence and can be easily optimized for memory layouts. However, the 1000X speedup discussed here does not cover the entire application and ignores many factors that limit total speedup due to Amdahl's Law effects.

Neutron Transport. Nelson (2009) in his thesis shows a variety of models and considerations for his performance results. His work solving neutron

transport considered multiple models for running the problem and optimizing for the GPU. The model that produced his best results shows 19X from a 49,152 neutrons per batch run for single-precision. The same model shows 23X when using single-precision and fast math. For double-precision performance the fastest speedups he observed were 12X.

Work done by Gong et al. (2011) in a MCNP-based application has similar performance benefits to Nelson's work. Speedup factors of 16X to 23X were found depending on problem parameters. This work was only an introductory attempt at implementing MCNP in CUDA, as MCNP is so large that it is time intensive to consider more than a subset of possible features and problem types.

Heimlich, Mol, and Pereira (2009) reported a speedup of around 15X for his neutron transport application when comparing a GPU to an 8-core CPU. This work focused on optimizing a history-based approach in CUDA for the GPU and using MPI+OpenMP for the CPU. This particular algorithm contained only small amounts of divergence in the code path that computes the random walk of neutrons, providing a possibility for greater use of available parallelism.

Gamma Ray Transport. Work presented by Tickner (2010) on X-ray and gamma ray transport uses a slightly modified scheme from the others by launching particles on a per block basis. In this way, he hoped to remove the instruction-level dependencies between particles running on the GPU. In this work, he produced speedups of up to 35X over a single core CPU, a significant improvement over similar methods launching with a particle-per-thread scheme.

Coupled Electron Photon Transport. Jia et al. (2010) discussed work in a dose calculation code for coupled electron photon transport that follows a relatively straight-forward algorithm. In their work, they offload the data and

computations to the GPU, simulate the particles, and then copy memory back. This method produced a modest performance increase on a GPU of around 5 to 6.6X over their runs on a CPU. The limitation of this speedup was attributed to the branching of the code.

Track Repeating Algorithm. In contrast to Jia et al.'s work, Yepes et al. (2010) showed that a different algorithm could greatly improve results. By converting a track-repeating algorithm instead of a full physics Monte Carlo code, Yepes et al. gained around 75X the performance on the GPU over the CPU. It is thought that the simpler logic of this algorithm generated threads which followed less branching paths than the algorithm presented in Jia et al.'s work.

Performance Evaluation. All of these examples contain a common theme. While performance can be gained doing Monte Carlo on the GPU, it can be more difficult to get than expected due to the highly divergent nature of the Monte Carlo algorithm. Methods to deal with this divergence show promising results. These outcomes are expected since Monte Carlo applications are embarrassingly parallel (good for GPUs) but also incredibly divergent (bad for GPUs).

In this section, we have seen a wide range in performances, from as low as 5X to as high as 75X, or even 1000X. While simplifications played a large role in the 75X algorithm we do see a full Monte Carlo application achieving speeds of 35X in the case of the work by Tickner (2010). It is important to note that while some of the differences in performance are due to the nature of each problem being solved, the algorithmic choices made can have a significant impact on the GPU implementations.

Algorithms. Based on the preceding performance studies, it is important to highlight the algorithmic approaches that were taken in order to understand the

performance of each approach. Clearly identifying algorithms that show positive performance results has the potential for gain in other codes. Therefore, this section surveys several of the important algorithms.

Monte Carlo transport applications tend to follow a simple model where each tracked particle is given its own thread and computation progresses in an embarrassingly parallel fashion. On a GPU, this also makes sense as a starting point since particles are independent and this progression leads to a naturally parallel approach. It is often pointed out, however, that due to the divergent nature of Monte Carlo this approach might not be the best way organize Monte Carlo codes on GPU hardware.

Particle-Per-Block. We will first look at an alternative approach, the particle-per-block tracking algorithm described by Tickner (2010). First each tracked particle or quantum of radiation is given to a block of threads. Then calculations are performed for one particle on each block of threads. For example the particle intersection tests with the background geometry can be performed in parallel on those threads for each piece of geometry that particle might be able to collide with. Areas where these parallel instructions can be utilized within a particle's calculation are then used by the threads in a block computing for that particle.

This particle-per-block technique is effective in mitigating the divergence issue. Particles often diverge quickly from one another in the code paths they follow. This means that threads in a block are not always able to travel in lock step and can cause some serialization of the parallel regions. By using only one particle per block, the divergence problem is nearly removed from the equation. Additionally, this method introduces a new area of parallelism that is not otherwise

being taken advantage of: instruction-level parallelism in the calculations for a single particle.

This method, however, does not take full advantage of the parallelism in the hardware like the methods that are not sensitive to divergence. Many threads can execute simultaneously within a block with potential slowdowns coming from when groupings of 32 threads are held in a warp and forced into the lockstep pattern. Running only one particle per block can sacrifice some parallelism, as not all tasks to calculate a particle's path are parallel operations. Additionally, since warps are scheduled out of thread blocks, any particle operations that are not done in parallel among the threads of a block are serializing themselves in a similar manner as to those algorithms that run one thread per particle and contain high levels of divergence.

In summary, this method has some merit if it can find enough parallel work in the thread block to execute additional parallel tasks that would otherwise be stalled when following a simpler method. Also, this method might end up showing the same characteristics of the simpler particle-per-thread model if the extra parallelism is not found, and instead lose out on the parallelism provided by particles that are not highly divergent from one another.

Event-Based Approaches. A second, possibly more obvious method, to escape the divergence issue is to switch particle tracking algorithms more dramatically from a history based version to an event based version. Subsection 2.4.4, *Event Based Techniques*, discusses this topic in more detail. Event based approaches require much more work then simply transforming an existing code to use the history based version on the GPU. Four separate works — Xu et al. (2015) X. Du, Liu, Ji, Xu, and Brown (2013) Liu, Xu, and Carothers (2015)

Table 2. GPU speedup evaluation results by Ding et al. (2011) for different Monte Carlo transport approaches

Case	Execution Time T_{CPU} (minutes)	Execution Time T_{GPU} (minutes)	Speed-up Factor T_{CPU}/T_{GPU}
Neutron Transport Problem	0.496	0.017	29.2
Eigenvalue / Criticality Problem	4.25	0.5	8.5
Voxelization	2380.4	52.3	45.5

Su, Du, Liu, and Xu (2013) — on the Monte Carlo code Archer, show that it is not simple to get speedups using this method as there are a host of challenges to still overcome. These challenges include: additional levels of divergence, atomic operations, data locality/layouts, and portability.

Evaluation. A number of studies were conducted by groups identifying the potential benefits of GPU hardware but also software development issues with Monte Carlo applications. Among these concerns are memory limitations, lack of ECC (error correction code) support in memory, lack of software optimization, limitations of SIMD architecture, clock speeds, and complex memory allocation schemes. In addition, the achieved performance often did not exceed that of unchanged codes on a cluster. In some cases, though, speedups were large and easy to achieve, such as the approach from Ding et al. (2011). Their evaluations are

listed in Table 2, including a 45× speedup for their voxelized approach. The only strong conclusion from these works are that a clear and defined path are not yet known on how to take full advantage of the available parallelism without suffering performance penalties in turn.

2.4.2 Monte Carlo and Medicine. Monte Carlo transport in the area of medicine often gets overlooked by Monte Carlo practitioners. Radiation transport calculations are used for dose estimations in patients and require close to real time, highly accurate solutions on desktop style machines. The following are descriptions from three applications of medical Monte Carlo transport followed by an evaluation of the effect GPUs have had on the field.

Electromagnetic Monte Carlo transport in GMC. Jahnke, Fleckenstein, Wenz, and Hesser (2012) described his group's efforts to develop the code named GPU Monte Carlo (GMC). GMC is a GPU implementation of the low energy electromagnetic portion of the Geant4 code using the CUDA interface. GMC runs in a thread per particle style operating on 32768 particles at a time (128 blocks of 256 threads). GMC runs through a series of kernel launches in a loop each handling one important aspect of the physics.

The raw performance differences between the CPU version and the GPU implementation are significant for the problems tested. The average for their study showed the GMC histories being computed at a rate of 657.60 histories every milli-second compared to the Geant4 CPU with histories computed at 0.137 histories per milli-second. Comparing these two numbers produces a speedup factor for the particle tracking portion of 4860 while maintaining reasonable accuracy in all cases between CPU and GPU with accuracies greater than 95% in all regions. Total

runtimes were also brought down to the hundreds of seconds showing the possibility for clinical usage of applications like this.

Proton Therapy in gPMC. Accurately computing radiation doses is a critical part of proton radiotherapy, and Monte Carlo simulations are considered to be the most accurate method to compute those dose calculations. Given the long time required for traditional applications to use this technique, clinical application have been severely limited. Jia, Schümann, Paganetti, and Jiang (2012) describes a fast dose calculation code, gPMC, and how it might enable clinical usage of Monte Carlo proton dose calculations.

The code gPMC was developed in CUDA for use on a GPU. Using a batching system to launch groups of particles from a particle stack, gPMC runs for between 6 and 22 seconds to generate passing rates between 95% and 99%. Jia et al. (2012) explain that they have successfully developed a dose calculation code under a certain set of restrictions and are hopeful that their future work will be able to meet with continuing success as they expand the context for their application.

Electron-Photon Transport in DPM. Jia et al. (2010), as noted in Section section 2.4.1 (**Coupled Electron Photon Transport**), describes the development of a CUDA based Monte Carlo coupled electron-photon application for dose planning, called DPM (dose planning method). Their scheme involves launching a kernel on the GPU that simulates all of the particle histories necessary to reach some target number of source particles. Each thread of their kernel simulates the history of one source particle and all secondary particles that it generated. The kernel ends with an atomic gathering of all the dosing data. DPM

was only able to achieve speedups of around 5-6.6x on the GPU over the CPU, but did get excellent agreement on relative uncertainties in their results.

Jia, Gu, Graves, Folkerts, and Jiang (2011) revisits their DPM code and are able to improve speedups of 5-6.6x into speedups of approximately 69 - 87x. DPM's main algorithm changed in a few significant ways. First a single thread only computed the history of a single particle and any additional particles were placed on a stack for a future iteration. Second the photon and electron physics were separated into different kernels so that threads would experience less divergence when handling the necessary code paths. Other factors such as a better random number generator and use of the hardware linear interpolation features were also implemented. With the additions of new features and improvements, DPM re-evaluated their accuracy and found that their errors were not statistically significant in over 96% of regions for all problems they tested. Given the now excellent speedups of 69-87x and acceptable accuracy ranges, real time speeds for realistic problems was achieved.

Evaluation. These three projects show a variety of challenges and accomplishments in the medical Monte Carlo field. They are each accomplishing their tasks on a single GPU as opposed to a cluster of CPUs. There are numerous stated benefits to this, with cost of purchasing and operating a cluster against purchasing a single GPU being a large factor. In each case speedups were achieved that were adequate to bring the time of their simulations down to those that would be useful in a clinical environment.

2.4.3 Monte Carlo and Ray Tracking.

One important and often computationally expensive aspect of Monte Carlo transport is the step that determines if the particle will collide with any background geometry, or at least

cross into a zone of a different material. This is done through a similar method as that used in ray tracing. *Ray tracing (graphics)* (2021) is a technique in computer graphics for "generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects."

The general process of ray tracing is very similar to Monte Carlo transport in the need to do many intersection tests from potentially scattered sources. Bergmann (2014) decided to study the potential of using the power of a highly optimized GPU ray tracing library, OptiX as described by *OptiX Programming Guide* (2018). Parker et al. (2010) explains that OptiX is a scalable framework for building ray tracing applications used on NVIDIA hardware.

The first study conducted was to determine the optimum configuration for OptiX as well as the capability for OptiX to be initialized with random starting points and directions as is most likely to be the case in a Monte Carlo application. When using a ray tracing library it is important to consider the two areas that can scale: the number of concurrently traced rays and the number of geometrical objects in the scene. Bergmann (2014) explains that nuclear reactor simulations might contain thousands of material zones in complex geometric layouts; knowing this last scaling parameter is especially important to not overlook. In these studies, the rates became fairly consistent after reaching 10^6 particles. Bergmann (2014) also notes some important points, such as which acceleration structure was always best and when memory become a constraint on the problem that could be run. The conclusion from this study was that OptiX could be used to handle the geometry representation in a Monte Carlo neutron transport code. Additionally, for best performance one should use a primitive-based geometry instancing method, a BVH acceleration structure, and run as many parallel rays as possible.

In addition to the use of a pre-existing tool like NVIDIA's OptiX library, other groups looked at optimizing Monte Carlo transport by focusing on treating it like a ray tracing problem. Xiao, Chen, Hu, and Zhou (2015) focused on the data locality issues in all ray tracing applications on GPUs. They describe a new data locality method based on task partitioning and scheduling in order to enhance spatial and temporal data locality by ordering random rays into coherent groups. By applying this method they achieved a 6-8X speedup over the previous GPU version of radiation therapy Monte Carlo transport. Després, Rinkel, Hasegawa, and Prevrhal (2008) studied the ray tracing algorithm for tracing a path through a grid in the context of Monte Carlo applications. Their GPU implementation of the Suddon algorithm, showed a speedup factor of 6X over the CPU. This work provides context for an important portion of the Monte Carlo transport problem, a look at the transport piece itself.

These examples show that progress in connected fields can positively impact the approaches in Monte Carlo transport. Ray tracing is only one aspect of a full Monte Carlo transport application but it can be greatly beneficial to look at work done in these related fields and bring those ideas back into the full application.

2.4.4 Event-Based Techniques. Much discussion has been aimed at the negative effect divergence in Monte Carlo codes has on performance. Given the embarrassingly parallel nature of the Monte Carlo transport algorithm, performance of Monte Carlo transport codes on the GPU should be incredible. This survey has shown however that the opposite is often seen in practice. Many applications achieve only marginal speedups, citing that the cause of their lack in performance was due to divergence in the code.

In order to combat divergence and given the similarities between the classic vector machines of the 1980's/90's with modern GPU hardware, algorithms developed for vector machines were re-evaluated for use on GPU architectures. In particular, the event-based approach worked well on SIMD vector hardware. In the event-based approach particles are processed in groups which perform the same event. There are multiple variations to this idea, a few of which are presented here.

Vectorized Algorithm. Early event-based algorithms were designed for vector machines and were called vectorized algorithms. Martin (1989) describes a successful vectorized algorithm as well some variations. The conventional Monte Carlo algorithm cannot be vectorized since treating many histories simultaneously would immediately fail after the first step of the simulation as each particle can undergo a different event. In order to achieve vectorization the histories need to be split into events which are similar and can be processed in a vectorized manner, i.e., the same set of instructions. The basic event-based iteration algorithm is described in Algorithm 3.

In addition to the basic event-based approach there are a few variations discussed in Martin's paper that expand on this model. One variation is the stack-driven approach. In this approach the events are further divided into smaller computational tasks. Instead of cycling through the tasks in a fixed order, the computation can move forward by selecting the event with the largest number of particles. This involves a tradeoff of simplified control flow versus maximizing the vector lengths of the computational components.

In recent work by Ozog, Malony, and Siegel (2015), multiple approaches to vectorization were tried. The banks of particles method described in Ozog et al.'s paper follows the same form as the original basic stack based algorithm, with sub-

Algorithm 3: The basic iteration event

for *event n = 0, 1, 2, ...* **do**
- Fetch Γ^n
- Perform free flight analysis:
  - gather the cross section data and geometry data tabulated by particle,
  - $\Sigma \leftarrow$ S,
  - $\rho \leftarrow$ R;
  - using Σ, sample a vector of distances to collision, d_c
  - using ρ, determine vector of minimum distances to boundary, d_b
  - determine the minimum distances to the end of event,
    - $d_{min} = \min[d_c, d_b]$;
  - update the particle coordinates,
    - $r^{n+1} = r^n + \Omega^n \cdot r_{min}$
  - Perform collision analysis:
    - gather particle attributes,
      - $\Omega \leftarrow \Gamma^n, E \leftarrow \Gamma^n$;
    - evaluate collision physics for new direction cosines and energies,
      - $\Omega' \leftarrow \Omega$, $E' \leftarrow E$
    - scatter new particle attributes back into bank,
      - $\Omega' \leftarrow \Gamma^n$, $E' \leftarrow \Gamma^n$
  - Perform the boundary analysis:
    - gather particle zone indices Z,
      - $Z \leftarrow \Gamma^n$
    - determine new zone indices,
      - $Z' \leftarrow Z$:
    - scatter new zone indices back into bank.
      - $Z' \rightarrow \Gamma^n$
  - Update the particle bank,
    - $\Gamma^n \Rightarrow \Gamma^{n+1}$ (*with* L_{n+1} *particles*)
    - (e.g. compress out terminated particles).
  - If $L_{n+1} \neq 0, continue$

stacks to manage vectorizable groupings of particles. A second idea, that offered performance benefits of 1.6x for an Intel many integrated core (MIC) processor over an Intel Xeon processor, was to vectorize nuclear data lookup portions of the code. In this way particles are processed in the same history based manner (little to no code changes required), and vector units are utilized to perform the expensive nuclear data lookup and interpolation calculations.

Vectorized versions of the Monte Carlo transport algorithms are generally based on this original basic algorithm. There are many variations but the principal differences all depend on the methods used for organizing and treating the vectors of particles. There are variations using stacks, tags, and tasks. When considering changing an existing history-based legacy code, the major downside to the event-based approach is that it requires large modifications to pre-existing source code.

Event-Based for GPU. Event-based methods used for the GPU follow similar design patterns as those that were developed for vector machines. One prime example is the event-based version developed by Bergmann (2014) for the code WARP. Figure 2 outlines the inner transport loop broken into its separate stages. Figure 3 outlines the outer transport loop between neutron batches.

Bergmann (2014) utilizes a series of kernels that each solve one piece of the process. Once each neutron knows which path it will go down – i.e. scattering, fission, etc. – each of those possible paths is launched in a separate kernel. Unlike the basic vectorized approach or the stack based approach however, all of the events are launched at once using concurrent kernels due to CUDA streaming properties. In this way, the main divergent part of the code is broken into relatively non-divergent kernels which are then launched simultaneously so as to continue to utilize the full hardware.

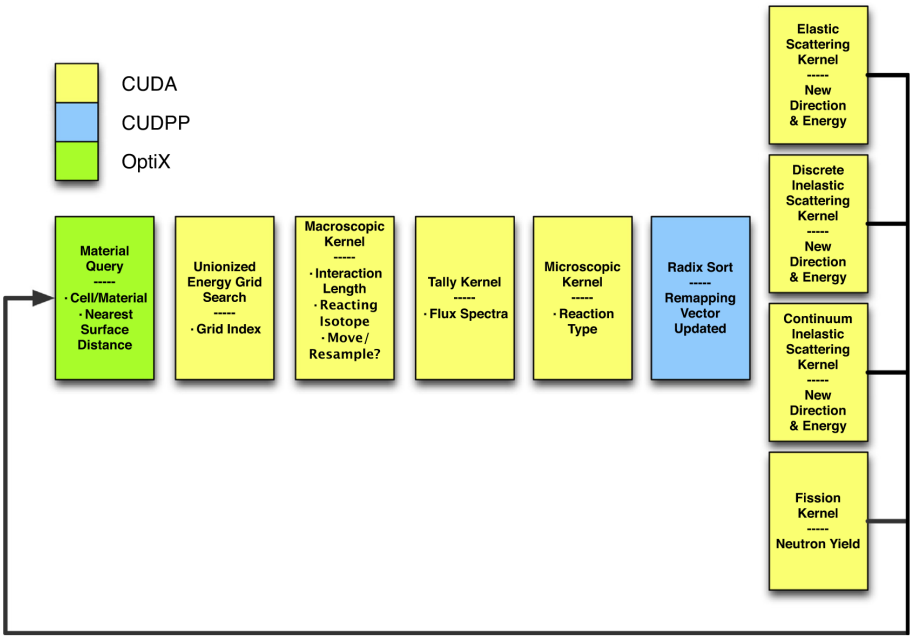


Figure 2. WARP inner transport loop that is executed until all neutrons in a batch are completed Bergmann (2014)

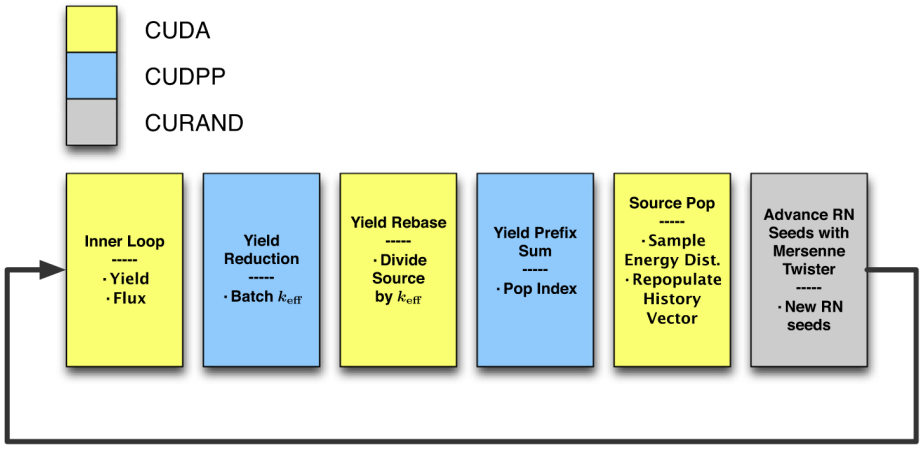


Figure 3. WARP outer transport loop that is executed in between neutron batches for criticality source runs Bergmann (2014)

Not all attempts at vectorization, or implementing an event-based algorithm for Monte Carlo transport codes, have been successful. For example, Liu, Du, Ji, Xu, and Brown (2014) describe an event-based approach that produced a roughly ten times slower version then the history-based code. This example shows how complicated the task of implementing an event-based algorithm can be, and that it is possible as well that not all Monte Carlo transport problems can be solved efficiently in an event-based fashion. Liu attributed their slow down to the memory access latency due to the high amount of global memory transactions and showed that the resulting cost of this in an event-based method did not outweigh the benefit of reducing thread divergence and increasing warp execution efficiency.

Continuing Work. This dissertation contains results that continue this work. Chapter III, Tracking Algorithms, gives a detailed account of our first attempts at event-based Monte Carlo in a research mini-app, ALPSMC. Bleile, Brantley, Dawson, et al. (2016) and Bleile, Brantley, O'Brien, and Childs (2016) give a full accounting of this work as well.

More recent work by Hamilton, Slattery, and Evans (2018) expand upon this idea further. They explain their implementations for history- and event-based algorithms in the Profugus Monte Carlo code. Additionally, they showed that with performance optimizations in their history-based approach, targeting reducing thread level divergence, their history-based approach out performed the event-based approach by 3 to 7×.

Hamilton and Evans (2019) followed up this work by implementing these algorithms with some additional improvements in the Monte Carlo code Shift using a continuous energy model instead of multi-group energy model. Their findings in this work contradict their previous work showing that the event-based approach

was a significant improvement when performed in Shift. The reason for this is attributed to the increased complexity of Shift, increasing register usage and limiting occupancy when running history-based. Since the event-based method targets smaller kernels with less registers the higher occupancy is attributed for the performance gains. This work shows a significant result, that the mini-app development for this topic might not be representative enough of the full application for the results to translate to the production application. Because of these results, we will further investigate some initial event-based results in this dissertation in Chapter VII Performance at Scale.

2.5 What is Portable Performance

Wolfe (2016) defines portable performance as the ability to achieve a high level of performance on a variety of architectures. In this case, high performance is relative to each target system. One important consideration, then, is the target architectures.

While it is clear that there will be an increase in node-level parallelism, it is unclear which specific many-core architectures will be used on future supercomputing platforms. There are many different architectures and vendors to choose from when designing a supercomputer, and there is currently no consensus for a single choice among the many options. Currently GPU based systems are a leading choice, but even among that distinction there are a number of contending vendors, leading to a wide variety of platforms and ideas to consider in this space. For example, NVIDIA provides General Purpose Graphics Processing Units (GPGPUs) which are highly parallel throughput-optimized devices, and the Summit and Sierra supercomputer procurements are IBM+NVIDIA based systems as defined by the *Fact Sheet: Collaboration of Oak Ridge, Argonne, and Livermore*

(CORAL) (2014). Additionally, back in 2015, Los Alamos National Laboratory (LANL) chose an Intel Xeon (Haswell) and Xeon Phi based system as described in the *About Trinity* (2015) article. Continuing in this may-core trend Thomas (March 2020) explains LLNL's recent announcement to partner with HPE and AMD on their next supercomputer procurement, El Capitan.

Application developers now face a complex and varied path forward. There are additional levels of complexity and potentially large changes to designing simulation codes in order to effectively utilize this increase in parallelism. In addition, the factors behind supporting a new architecture are often more complex in the context of legacy codes and/or codes that aim to run effectively on many architectures. The simulation code developer must now address both the issue of portability and the issue of performance of their algorithms, or risk their simulation code becoming outdated or unusable very quickly. This problem is especially challenging when optimizations are specific to one architecture (i.e. not portable across platforms). Given this wide array of possible architectures, the value of portable performance has never before been higher.

2.5.1 Portable Performance Applications. In this section we will look at the meaning of portability and the uses of portable performance abstractions in different contexts. Moreland, Larsen, and Childs (2015) discussed the need for portable performance solutions for visualization software in their paper, "Visualization for Exascale: Portable Performance is Critical." While their context is visualization, their arguments span more than just visualization applications. Current scientific, legacy applications will not be adequate to run on exa-scale machines, or even the recent peta-scale machines. Further, applications such as these need to be ready to run on new machines as they arrive. Therefore,

these applications should strongly consider portable performance solutions in order to maintain relevance in the upcoming computing environments.

Portability does not mean the same thing to everyone. In some cases, portability might be as simple as requiring minimal code changes to run on a new architecture. Bosilca et al. (2011) describe portability for their application as requiring no code changes to the main body of code in order to enable GPUs. In this work, the only allowable changes were those in CUDA to launch kernels while the code that made up the kernels had to remain unchanged. In this work they added a scheduler that could launch code regions onto different platforms based on availability and which one is the most beneficial for performance.

P. Du et al. (2012) describe their work as portable since they transitioned from CUDA to OpenCL. The goal of their work was to understand the performance tradeoff between a more portable OpenCL implementation of an algorithm and the vendor specific GPU language CUDA. In this work Du et al. showed that the performance of OpenCL was similar to CUDA for the compute intensive kernels but also had a higher overhead. Also, it is important in OpenCL to account for architecture specific features or designs for optimum performance.

Portable performance as was laid out in Moreland et al.'s paper can be seen in recent work done in the field of ray tracing and volume rendering. Larsen, Meredith, Navrátil, and Childs (2015) presented his work for a method of ray tracing consisting entirely of data parallel primitives. In this work, all parallel operations are expressed using data parallel primitives, which are defined in a library. The definitions of these primitives are then compiled to be CUDA, OpenMP, or serial executions. The performance of this method is shown to be competitive with both of the top ray tracing libraries, OptiX from NVIDIA and

Embree from Intel. In addition to ray tracing Larsen, Labasan, Navrátil, Meredith, and Childs (2015) followed up this work with a volume rendering capability that uses the same principles for portable performance. This work shows volume rendering performance similar to current industry standards while also maintaining portability.

Portable performance also exists outside of the visualization realm with scientific codes also utilizing the new methods being developed. Rahaman et al. (2015) presents work on portable performance for nuclear reactor models by using the OCCA programming model. OCCA, or the Open Concurrent Computing Abstraction, provides an interface into parallelism that can be compiled for different architectures. Like many of the abstraction layers that will be discussed in Section subsection 2.5.2 (Abstraction Layers), OCCA provides the ability to write code once and compile it into different known formats such as OpenMP, CUDA, or serially. Rahaman et al. compared the performance of this abstraction against native implementations in order to weight the usefulness of the portability it provided. Their results showed that, for some cases optimal performance could be achieved simply on both a CPU and a GPU, but in other cases it took complex specialization in order to make the same kernel work well on both architectures.

Portability has become a pressing point for developing applications in today's computing environment. Applications like those Rahaman studied, which performed less than optimally on some architectures, are quite common as architecture specific optimizations become increasingly difficult to balance. An added complication to this issue is the inclusion of legacy codes into the mix. Since legacy codes are already developed and might require massive rewrites to take on

a new architecture like a GPU, finding a portable performance solution will be critical to their future development.

2.5.2 Abstraction Layers. Abstraction layers are a frequently used method for achieving portable performance. The key idea behind an abstraction layer is to hide the complexity of parallelism behind an abstraction. Then the abstraction can handle how to parallelize a given section of code onto separate hardware architectures. The remainder of this section surveys some of the known abstraction layers with a summary of their benefits and goals.

OpenMP. Parallelism through *OpenMP* (2018) is achieved through the use of compiler directives, library routines, and environmental variables. These are used to specify the high level parallelism for programs using the Fortran and C/C++ languages. These directives, routines and variables have been expanded to include methods to describe how regions of code or data should be moved to another computing device, like an accelerator.

S. Lee, Min, and Eigenmann (2009) describe several advantages for using OpenMP as a programming paradigm for use on a GPGPU:

- "OpenMP is efficient at expressing loop-level parallelism in applications, which is an ideal target for utilizing GPU's highly parallel computing units to accelerate data-parallel computations."
- "The concept of a master thread and a pool of worker threads in OpenMP's fork-join model represents well the relationship between the master thread running on the host CPU and a pool of threads in a GPU device."
- "Incremental parallelization of applications, which is one of OpenMP's features, can add the same benefit to GPGPU programming."

Ayguadé et al. (2010) explains that by including target device directives as well as other supporting features, OpenMP is able to utilize its experiences in parallel computing and offer a familiar solution to programmers who need to make new or existing algorithms and codes work for parallel CPUs, GPUs. Additionally, S. Lee and Eigenmann (2010) expands upon this showing that OpenMP can be extended to improve GPU specific performance through additional tunable parameters.

OpenACC. *OpenACC* (2018) enables the offloading of loops and regions of code onto accelerator devices. The OpenACC API uses a host-directed model of execution where the main program runs on the host, or CPU, and the computational work is offloaded to a device accelerator, like a GPU. The OpenACC memory model outlines two memory spaces which do not automatically synchronize, requiring explicit synchronization calls between memory spaces. Wienke, Springer, Terboven, and an Mey (2012) explain that OpenACC operates in a similar fashion to OpenMP by using compiler directives to define regions of code for their operations to affect.

OpenACC (2018) is designed to be portable. Its directive based programming allows programmers to create high-level host+accelerator applications without needing to explicitly handle many of the extra aspects to working on an accelerator.

OpenACC has demonstrated the ability to achieve reasonable performance on multiple platforms. Wang, Qin, SEE, and Lin (2013) performed a performance study showing that for some benchmarks the OpenACC versions were able to achieve more than 82% performance when compared with peak performance for both the Intel Knights Corner and NVIDIA Kepler architectures.

Thrust. *Thrust - Parallel Algorithms Library* (2018) is a library of algorithms and data structures that can be used to provide an interface to parallel programming in order to increase a programmer's productivity. Thrust is designed similar to the standard template library, allowing programmers familiar with the C++ STL to feel instantly comfortable working in the Thrust environment. Hoberock and Bell (2010) explains that through its design, Thrust lowers the barrier to entry for allowing access to GPU hardware and memory without the need to interact with the CUDA API.

In addition to adding parallel algorithms, Thrust provides multiple compilable backend technologies that allow the programmer to write their algorithms using Thrust and then compile them in CUDA, TBB, and OpenMP. This enables a wide array of portable solutions that programmers can take advantage of in order to much more easily write portable and performant applications.

Thrust offers a variety of algorithms with significant performance advantages to direct naive implementations, leading to real world performance gains. Some examples of those performance gains can be seen in the implementations of the fill and radix sort algorithms. Bell and Hoberock (2011) explain how Thrust provides a fill algorithm that produces a 32x performance gains over a naive algorithm implementations as well as a radix sort algorithm that provides a 2.7x performance gain by utilizing only significant bits when possible. These performance gains come for free when using a Thrust algorithm to accomplish a data parallel task.

In addition, Thrust provides all of the main data parallel operations defined by Blelloch (1990). Blelloch's work is significant in that it provides a foundation for data parallel processing and algorithm development through a series of well defined

vectorized operations. One method of achieving performance is to then rewrite an algorithm using data parallel primitives or algorithms and then use the existing Thrust methods to perform the operations.

RAJA. The *RAJA* (2021) portability layer is designed to be a lightweight method of providing loop-level parallelism in existing codes. The idea behind the design was that, especially at institutions like Lawrence Livermore National Laboratory, there are a large number of legacy scientific codes that will need to make some sort of transition in order to utilize upcoming architectures. RAJA was designed to be able to replace current loops with a wrapper loop to at first make no change or impact. Hornung, Keasler, et al. (2014) describes that once the RAJA abstraction layer is in place, the loop can be changed to run on different architectures and with different parallel modes.

RAJA achieved their flexibility through macro replacements in their library. By changing a compile time option the user can define if they want the OpenMP parallel launcher, a CUDA kernel launcher, or a serial launcher. Hornung, Keasler, Kunen, Jones, and Beckingsale (2016) explain that, in this manner RAJA is a useful tool for generically replacing large numbers of parallel loops with a consistent theme that creates inlined parallel code for the compilers to optimize, instead of large and sometimes convoluted template models.

In addition to providing a library, the RAJA project provides a second approach to portability. A RAJA like approach involves simple custom macro definitions, such as making a parallel loop by replacing a for loop with a macro function. Then different parallel launchers can be defined for each target architecture without changing the body of the loop, minimizing code redundancy between versions. Finally, at compile time one of the architectures or versions

of the parallel loop is chosen and all of the loops defined with the macro will be launched for whichever version was chosen.

Kokkos. The *Kokkos* (2021) C++ library provides a programming model that enables performance portability across devices. The objective of the Kokkos library is to allow as much of the user's code as possible to be compiled for different devices, while obtaining the same performance as a variant of the code that was written specifically for that device. Kokkos uses the idea of execution and memory spaces to provide an abstraction to the problem. In their model, threads are said to execute in an execution space, while data resides within a memory space. Edwards and Sunderland (2012) explains that the relationships are defined between the different execution and memory spaces.

Parallelism in Kokkos comes from parallel execution patterns; data parallel and task parallel patterns are used. The primary data parallel patterns are: parallel_for, parallel_reduce, and parallel_scan. The data parallel computational kernels are implemented as standard C++ functors.

Edwards, Trott, and Sunderland (2014) demonstrated that the Kokkos abstraction layer can achieve 90% of the performance of optimized architecture specific versions for kernel tests and mini-applications. Additionally Edwards, Sunderland, Porter, Amsler, and Mish (2012) demonstrated Kokkos performance on Xeon, Xeon Phi, and Kepler architectures, showing the portability of this solution.

Chapel. Sidelnik, Maleki, Chamberlain, Garzar'n, and Padua (2012) described Chapel as an object-oriented parallel programming language which was designed from first principles. Chapel was developed in order to improve the programmability and productivity of development on parallel machines. B. L. Chamberlain, Callahan, and Zima (2007) defines productivity as "a

combination of performance, programmability, portability, and robustness." Chapel used this idea to make a global-view parallel language that uses a block-imperative programming style. Chapel purposely avoided building on the C or Fortran languages in order to help programmers avoid falling back into sequential programming patterns.

Chapel uses a code generation design to generate parallel C or CUDA code. The Chapel language defines the parallelism and so can be used as the basis for optimized code generation on many different platforms. Chapel uses this design to achieve portability and performance with their language.

B. Chamberlain (2013) describes that Chapel's design goal is to support any parallel algorithm that a programmer could conceive without the need to fall back to other parallel libraries. Chapel supports concepts for describing parallelism separately from those used to describe locality. It supports programming at higher and lower levels, as well as providing advanced higher-level features such as data distributions or parallel loop schedules.

VTK-m. VTK-m is the result of a collaboration between three separate groups and three separate national labs coming together and joining forces with Kitware, the primary maintainers of the current VTK (Visualization ToolKit) software. Visualization applications use VTK in order to express visualization algorithms and data structures in their codes. VTK-m came about from the three projects, EAVL, DAX, and PISTON, with the design goal of being a portable performance solution for visualization applications and algorithms.

The VTK-m framework takes the concepts of data parallel primitives and patterns generated from those primitives to provide a framework for accomplishing visualization algorithms. These data parallel primitives can be compiled for

different platforms, helping VTK-m achieve portable performance as described by Moreland, Sewell, et al. (2016) and Moreland et al. (2015).

The contributions of the three projects to VTK-m are as follows:

- EAVL – Provided a robust data model.
- DAX – Provided a model for parallel work dispatching.
- Piston – Provided many data parallel algorithms and implementations.

EAVL. Meredith, Ahern, Pugmire, and Sisneros (2016) define EAVL, or the Extreme-scale Analysis and Visualization Library, and describes how it was developed with three goals in mind:

- A flexible data model – "Expanding on traditional models to support current and forthcoming scientific data sets."
- High parallel efficiency – "Improve memory and algorithmic efficiency through the enhanced data model, and support stricter memory controls and accelerator device memory models."
- Scalability – "Support distributed and data parallelism, and transparently target heterogeneous systems."

Dax. Moreland, Ayachit, Geveci, and Ma (2016) define Dax, or Data Analysis at Extreme, as a library developed to support fine grained concurrency for data analysis and visualization algorithms. This library provides a dispatcher that schedules worklets onto data items. Additionally, Moreland, Ayachit, Geveci, and Ma (2011) describes how the Dax toolkit simplifies the development of parallel visualization algorithms and provides a data parallel framework for scheduling and launching parallel jobs.

PISTON. Lo, Sewell, and Ahrens (2012) define The Portable Data-Parallel Visualization and Analysis Library, referred to as *PISTON, A Portable Cross-Platform Framework for Data-Parallel Visualization Operators* (2016), as a cross-platform library that provides operations for scientific visualization and analysis. These operations are performed using data parallel primitives and the NVIDIA Thrust library. PISTON uses Thrust to perform the data parallel operations and for its cross-platform compatibility. PISTON adds useful algorithms for data visualization and analysis as well as an interface into the Thrust calls.

2.6 Research Gaps

Understanding what research has been performed already is of vital importance to understanding and finding what gaps exist in the current available knowledge. Through the research presented in this chapter, we can see that Monte Carlo transport applications are both important and complex. We have seen that supercomputing platforms are changing, with an increased emphasis on on-node parallelism through many-core architectures. Additionally, we have seen numerous attempts to make Monte Carlo transport applications run on these architectures and struggle to gain performance along the way. This work clearly highlights the complexity of the space as works often contradict, or are forced to focus on simplified sub-problems to get performance.

Throughout all of the work we can see that there are still a gaps in understanding how to ensure Monte Carlo transport can run effectively on these new architectures. In Chapter I, section 1.1 (Research Questions), we state the topics of interest that the remainder of this dissertation will follow. Firstly, the question of whether to use a history-based approach continues to vary as multiple groups have found conditions and applications where one or the other is more

performant. Secondly, data management through atomics or replications is often not addressed on the studies presented and clarifying this point will make it easier for others to continue exploring this space without repeating the same efforts, or running into unforseen pitfalls. Thirdly, the idea of utilizing the entire node is not addressed in any of the works we have reviewed, at least not in the context of using both CPUs and GPUs for computation at the same time. Finally, all of the work that has been in mini-apps needs to be evaluated in production applications in order to ensure that our understanding is complete, and not simply a product of simplifications made when developing the mini-app.

CHAPTER III

TRACKING ALGORITHMS

The work in this chapter is a combination of two previous publications. The first publication is in volume 114 of the Transactions of the American Nuclear Society in Summer 2016. Dr. Patrick Brantley initially identified the need for this work and provided the application that this work was performed in. I was the primary contributor to this work in developing the algorithm, writing the new code, and writing the paper. Dr. Patrick Brantley, Shawn Dawson, and Dr. Matthew O'Brien provided ideas and feedback throughout the development process and assisted in editing the paper. Dr. Hank Childs assisted in editing the paper. The second publication is in volume 115 of the Transactions of the American Nuclear Society in Winter 2016. This work is an extension of the previous work with the same division of labor.

3.1 Introduction

In this chapter we present our investigations into event-based Monte Carlo tracking algorithms and compare them with a traditional history-based approach. For this research, we began with the ALPSMC Monte Carlo test code P. S. Brantley (2011) that models particle transport in a one-dimensional planar geometry binary stochastic medium. This chapter is divided into two parts. In part one we explore an initial implementation of the history and event-based algorithm and report on our findings. In part two we develop optimizations for both our history and event-based approaches and then revisit our findings.

3.2 Part 1: Initial Implementation

This section presents the algorithms for history and event-based Monte Carlo transport. It then discusses the implementation details of our event based

algorithm. Finally, this section presents our initial findings, comparing our results with Thrust, CUDA, and a serial implementation.

3.2.1 History-Based Approach. The ALPSMC code was originally implemented in C++ using a standard history-based Monte Carlo transport algorithm, as shown in Alg. 4. This approach follows a single particle from creation until it is absorbed or leaked. Parallelism is easily added by parallelizing over particle histories (foreach loop on Line 1), with each thread working independently on a single particle at a time. In addition, ALPSMC is implemented using double precision floating point numbers throughout, which is required to attain sufficient accuracy.

Algorithm 4: History-based Monte Carlo algorithm

```
1  foreach particle history do
2      generate particle from boundary condition or source
3      while particle not escaped or absorbed do
4          sample distance to collision in material
5          sample distance to material interface
6          compute distance to cell boundary
7          select minimum distance, move particle, and perform event
8          if particle escaped spatial domain then
9              update leakage tally
10             end particle history
11         if particle absorbed then
12             update absorption tally
13             end particle history
```

3.2.2 Event-Based Approach. Previous researchers, as mentioned in the related work, have noted that the use of an event-based Monte Carlo particle transport algorithm such as by Brown and Martin (1984) may be beneficial for GPU or vector-based architectures. We investigated this idea through the event-based algorithm shown in Alg. 5 as a way to potentially optimize performance on

GPU and vector-type architectures. In event-based particle tracking, the individual events can be treated by a series of data parallel operations. The data parallel model matches the vector and GPU hardware with an emphasis on performing the same operations on many pieces of data at one time through SIMD (Single Instruction Multiple Data) parallelism. GPUs do not require full SIMD parallelism, but can benefit from it regardless.

Algorithm 5: Event-based Monte Carlo algorithm

```
1  foreach batch of particle histories (fits in memory constraint) do
2      generate all particles in batch from boundary condition or source
3      determine next event for all particles (collision, material interface
         crossing, cell boundary crossing)
4      while particles remaining in batch do
5          foreach event E in (collision, material interface crossing, cell
             boundary crossing) do
6              identify all particles whose next event is E
7              perform event E for identified particles and determine next event
                 for these particles
8          if particle escaped spatial domain then
9              update leakage tally
10         if particle absorbed then
11             update absorption tally
12         delete particles absorbed or leaked
```

3.2.3 Thrust. *Thrust* (2014) is a C++ header library using a STL-like template interface. Thrust provides a number of parallel algorithms and data structures designed to provide access to GPU computing without needing to write *CUDA* (2014) code directly. Additionally, Thrust provides backend capabilities allowing these algorithms and data structures to target different devices, including CPUs with OpenMP threads. This design was used for studying portable performance techniques with Thrust, providing a method of maintaining only one source code.

Thrust algorithms are used for implementing procedures across all particles in a batch. As discussed in Chapter II, these algorithms perform operations such as the data parallel map, reduce, gather, scatter, or scan operations defined by Blelloch (1990). Each of these operations can be performed in a data parallel way. Thrust also provides data types that can be used to manage memory for GPU devices. The thrust::device_vector and thrust::host_vector data structures operate similarly to a C++ std::vector but with automatic memory copying between host and devices whenever necessary. These data types allow for simple memory management schemes that work on both GPU and CPU based architectures.

3.2.4 Algorithm Detail. An event-based algorithm focuses on performing data parallel operations across all particles undergoing the same event. Additional overhead is needed to find the grouping of particles that will be operated on and to determine an access pattern for the particles. This reorganization stage can be costly and is not directly related to solving the transport problem.

Thrust provides permutation iterators that allow for the unaligned access of data elements according to an index map. Using this iterator scheme, data elements do not need to be copied into new locations for each operation. This approach comes at the cost of performing non-contiguous memory accesses for reading and writing the information.

In order to perform an event operation on particles using this scheme, a series of data parallel operations is used to establish the correct index mapping for the permutation iterator. This scheme is defined as follows and describes in detail lines six and seven of Alg. 2:

Step 1: thrust::transform — Fill out a stencil map of 1's and 0's of all particles doing event E (where each particle whose next event is E will get a 1 in the stencil map at its index location)

Step 2: thrust::reduce — Count the number of elements labeled 1 in the stencil (determines the number of particles that will perform event E)

Step 3: Check if the number of elements is greater than 0 (check if any particles are performing event E)

Step 4: thrust::exclusive_scan — generate indices for index mapping from stencil map (indices for each particle performing event E)

Step 5: Allocate a new map of appropriate size (map to hold indices for all particles performing event E)

Step 6: Scatter indexes from scan into new index map (reduces the exclusive_scan generated indices into the map that holds only enough for particles performing event E)

Step 7: Use new index map in permutation_iterator loops over all particles (combining the index map with the permutation iterator allows loops over all particles to operate only on the particles selected in the index map)

3.2.5 Implementations. We implemented the event-based version of ALPSMC using both the Nvidia CUDA programming model explicitly and the Nvidia C++ Thrust library. The Thrust implementation of ALPSMC utilizes data parallel operations and Thrust data types for managing memory. The same Thrust event-based implementation can be compiled with either CUDA for use on GPUs or OpenMP for use on CPUs, enabling portability to different platforms.

In the native CUDA implementation of ALPSMC, we found it useful to continue to use Thrust algorithms in building various maps, and used CUDA to directly launch event kernels instead of calling Thrust::foreach. The Thrust and CUDA implementations of ALPSMC give physics results identical to the original history-based implementation.

The CUDA implementations for this study matched the algorithm in the Thrust implementations. The differences in performance come from the capabilities that native CUDA programming provide that cannot be accomplished with Thrust. Using CUDA directly enables more fine-grained control at the kernel level and enables important access to different memory spaces such as GPU shared memory. The CUDA implementation includes a scheduling algorithm to optimize the number of active threads on the GPU for each kernel call. Additionally, the CUDA implementation includes the use of the different available memory spaces, such as constant and shared memory. For example, Monte Carlo particles were initially allocated in GPU global memory and then copied to shared memory for all operations within a kernel. All problem constants such as cross sections and mean chord length values were placed in GPU constant memory. These optimizations under certain conditions have a significant impact on the performance of a GPU kernel.

3.2.6 Initial Results. We performed scaling studies in which we varied the number of Monte Carlo particle histories (problem size) and the implementation methodology (Thrust or CUDA). The results presented are for Case 1a defined by P. S. Brantley (2011), with a spatial domain of 10 cm. We also examined the differences in performance on three different computer platforms. LLNL's Rzgpu computer has Intel Xeon Westmere-EP 2.8 GHz host cores with

Nvidia Tesla M2070 GPU device accelerators. LLNL's Max computer has Intel Sandy Bridge 2.6 GHz host cores with Nvidia Tesla K20X GPU device accelerators. The Tesla K20X GPU has improved double precision performance over the Tesla M2070. LLNL's Rzhasgpu computer has Intel Xeon Haswell 3.2 GHz host cores with Nvidia Tesla K80 GPU device accelerators. We did not implement an MPI variation of this code and were therefore only able to utilize approximately half of the computational power of the Nvidia Tesla K80s.

Our first study aimed to identify the speedups of our event-based algorithm when compared to the initial serial history-based implementation. We computed speedups over a serial calculation by dividing the wall clock time of a serial run of the history-based version of ALPSMC on the host core of the given machine by the wall clock time of the event-based version of ALPSMC running on both a single host CPU and the GPU device. The speedups obtained on each computing platform are shown in Table 3.

Table 3. ALPSMC event-based Monte Carlo GPU speedups over serial history-based version

	Number Particle Histories		
	10^6	10^7	10^8
CUDA (K20X)	5.90	11.88	11.91
CUDA (1/2 K80)	4.88	10.49	10.51
CUDA (M2070)	3.96	6.05	6.05
Thrust (K20X)	2.11	2.60	2.60
Thrust (1/2 K80)	1.77	2.17	2.17
Thrust (M2070)	1.42	1.64	1.63
Thrust OpenMP Event	2.54	2.15	2.22

For this test, the Thrust implementation produces speedups ranging from approximately 1.4 to 2.6. Therefore, while the Thrust library potentially provides an approach to obtaining a portable implementation, it does not produce the

significant speedups we would expect on the GPU hardware. For this test, the speedups obtained using the CUDA implementation of the event-based algorithm are significantly larger than those obtained using the Thrust implementation by up to over a factor of four. We attribute this improved performance to the fact that CUDA offers more control over the memory spaces available on the GPU (e.g. shared memory) when operating on large kernels that perform multiple read/write actions. Thrust does not offer such flexibility and manages the memory allocation internally. We conclude based on these preliminary investigations that a direct CUDA implementation is more efficient than a Thrust implementation for event-based Monte Carlo. Also, the speedups on the Max platform (Tesla K20X GPU) are larger than on the Rzgpu platform (Tesla M2070 GPU) by up to a factor of approximately two, presumably a result of the improved double precision performance of the K20X. Furthermore, the Rzhasgpu platform (Tesla K80 GPU) shows similar performance to the Max platform (K20X GPU); we can assume around twice the performance were we to modify the research code to fully utilize all of the available K80 hardware.

The same Thrust event-based code implementation was compiled with OpenMP for use on the host CPU, demonstrating the portability of the Thrust implementation. The scaling study was repeated on Rzhasgpu's Intel Xeon Haswell CPUs with OpenMP using 16 threads/cores. The CPU performs similar to the GPU when Thrust is used to gain parallelism, with speedups of approximately 2.2. Using 16 OpenMP threads, we would expect a significantly larger speedup for Monte Carlo particle transport. Since the same code base is used with Thrust on both the CPU and the GPU, we can see the potential that exists for a single code base on multiple platforms. For this particular example, however, significantly

higher performance is achieved using the native choice of the CUDA event-based implementation for the GPU.

We performed a second more extensive scaling study varying the number of particle histories for the Thrust and CUDA event-based versions and the serial history-based version on Rzhasgpu (Tesla K80 GPU). The results of the scaling study are shown in Figure 4. We can see that both the Thrust and CUDA event-based versions have significantly higher overhead than the serial history-based version at low numbers of particle histories. But at a higher number of particle histories (starting at approximately 10^5 particle histories), the event-based versions of the code begin outperforming the serial history-based versions. We also observe that the performance gains of the CUDA version over the Thrust version start to become significant at higher numbers of particle histories.

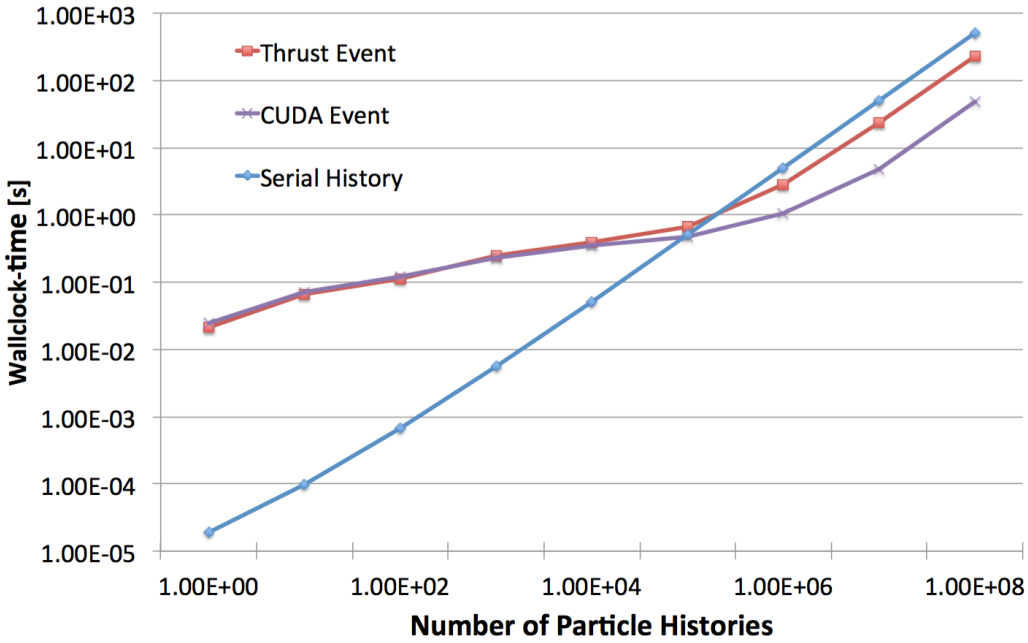


Figure 4. Log-log plot giving the timing for a scaling study comparing Thrust, CUDA and serial event-based approaches as a function of particle histories.

The linear behavior observed at the higher particle history counts is a result of the particle batching scheme we used to avoid exhausting GPU device memory. Once the batching begins, we no longer gain any additional performance increases. At that point, the only performance improvement possible would be to process a greater number of particle histories, and that number is hardware dependent.

3.3 Part 2: Optimized Implementation

In this section we highlight two important changes required to bring speedup to ALPSMC on GPUs. The first was to understand the effect of a data structure striding change, changing our particle data structure from an array of structures to a structure of arrays. The second was to understand the effect of particle removal schemes in our event based algorithm. Finally, with those changes, along with other minor modifications, we revisit our results from part 1 with new data.

3.3.1 Arrays of Structures Versus Structures of Arrays.

One way to gain performance on Nvidia GPUs is to coalesce global memory accesses in a streaming multiprocessor (SM) NVIDIA (2015). SMs schedule and execute threads in lock-step groups of 32 threads called warps. Memory accesses in each warp are coalesced in order to produce fewer memory transactions overall. For 16 threads in a warp, we can pull all 16 array values into the threads with a single call into global memory if we access the array in consecutive order. Since the threads in a warp operate in lock step, if memory accesses are not coalesced, more memory transactions are needed causing the threads in a warp to stall while more memory transactions are issued.

In order to accomplish coalesced memory accesses on larger data structures, such as the particle class used in the ALPSMC C++ Monte Carlo implementation P. S. Brantley (2011), a common recommendation is to transition

from an array of structures (AOS) data structure to a structure of arrays (SOA) data structure. This transition is important for all SIMD or vector architectures (not only GPU architectures — see Pharr and Mark (2012)) and so makes sense as a starting point for continuing optimizations in our ALPSMC study. For ALPSMC, this transformation requires that the member variables in the particle class are separated into different arrays of those members in a parent class. This entire process can be encapsulated in a higher level interface allowing for a compile time choice to be made for which data structure option to use: AOS or SOA. Maintaining flexibility in this type of option is important when running on a diverse set of hardware, where common and important optimizations on one set of hardware might lead to performance loss on another.

We encapsulated the entire AOS/SOA choice in a particle_vault data structure that allows access to its member variables through an interface. For the AOS case, the particle_vault data structure is simply a container holding an array of particles. Accessing a particle's member variables from the particle_vault requires getting a particle at an index and then grabbing its member, i.e., "$Get\_X(index)\{\ particle\_vault.particle[index].x\ \}$." For the SOA case, the particle_vault data structure now holds arrays for each of the particle's member variables. Accessing a particle's member variables from the particle_vault requires first choosing the member and then accessing a value from an index location, i.e., "$Get\_X(index)\{\ particle\_vault.x[index]\ \}$."

3.3.2 Optimized Particle Removal Scheme. One discovery we made while pursuing optimizations for the event-based algorithm was the significant percentage of time spent removing inactive particles from the particle list. (The event-based algorithm is described in detail in Ref. Bleile, Brantley,

Dawson, et al. (2016).) The algorithm originally collected all particles undergoing each of the events and then performed that event on the list of particles. The last stage of the original algorithm was to always remove inactive particles (e.g. particles that were absorbed), i.e. perform material interface crossing events, zone boundary crossing events, collision events, and then remove inactive particles. Table 4 shows the wall clock times of each event for an example problem (Case 1a P. S. Brantley (2011) with a spatial domain of 10 cm) with ten million particles for both the AOS and SOA data structure implementations. All simulations in this section were performed on the LLNL Rzhasgpu computer. We can clearly see that always removing inactive particles dominates the time spent processing the events. We conjectured that if we could minimize this removal function, even at the cost of increasing compute time, we might be able to decrease the overall time spent processing events.

Table 4. Wall clock times [seconds] for each event for a 10 million particle study using the CUDA event-based method.

	AOS Data Structure		
Event	Remove Always	Remove Never	Remove Half Size
Material Interface	0.50	4.13	0.60
Zone Boundary	0.81	4.79	0.90
Collision	1.14	5.55	1.35
Remove	2.83	3.15	0.88
Total	5.28	17.62	3.77
	SOA Data Structure		
Event	Remove Always	Remove Never	Remove Half Size
Material Interface	0.39	2.23	0.44
Zone Boundary	0.64	2.98	0.77
Collision	0.73	3.50	0.93
Remove	4.46	1.48	0.87
Total	6.22	10.19	3.01

In order to justify removing inactive particles at all, we investigated not removing particles to show that, while removing particles is an expensive operation, it is more costly to operate on the full list every time. There is still some amount of time spent in the Remove stage of the algorithm because it is necessary to check if all particles have completed processing, which is the end condition for the simulation. In light of our conjecture above, we also implemented an algorithm in which we only remove inactive particles when removing them produces a significant impact on the size of the list. Following numerical experimentation, we chose to perform the remove operation if the number of inactive particles to be removed is at least half the size of the list. As a result, the maximum number of times we perform the expensive removal operation becomes log(n), where n is the size of the list. As shown in Table 4, the "Remove Half Size" algorithm produces a 1.4X and 2.1X improvement in total wall clock time over the "Remove Always" algorithm for the AOS and SOA implementations, respectively. Finally, we observe that the SOA implementation produces a 1.3X improvement in total wall clock compared to the AOS implementation.

We also investigated replacing the Remove function with a sort function to understand the full effect on compute and remove times, both sorting each iteration and in the same remove half scheme described above. The sorting resulted in a significant slowdown for each method when compared to the most efficient approach: 84.9X slower when sorting each time and 3.2X slower when using the remove half scheme. Overall, increasing removal times (that includes the time for the sort) far outweighs the cost of decreasing the compute times. As a result, sorting is not effective, even when we include the new conditional sorting scheme.

3.3.3 Results Revisited.

All simulations in this section were performed on the LLNL Rzhasgpu computer that has 16 Intel Xeon Haswell 3.2 GHz host cores with 2 Nvidia Tesla K80 GPU device accelerators per node. GPU results are run in maximum batch sizes of 10 million particles on a given CUDA device, with multiple batches able to run on different devices at one time. Each Nvidia Tesla K80 appears as two devices, so there are four CUDA devices usable at a time.

3.3.3.1 Thrust and CUDA Event-Based Approach.

The main conclusion from our previous results was that the event-based Monte Carlo transport algorithm is viable on GPU architectures, but the use of the Thrust library portability abstraction resulted in a significant performance penalty. With the goal of reducing this performance penalty, we reimplemented the Thrust library version of the code (that could run on CPUs and GPUs) based on the the most efficient CUDA version that incorporated the algorithmic modifications described above as well as some additional minor optimizations. This section presents the results of this work.

Figure 5 shows the results of a particle scaling study performed with the optimized event-based CUDA version compared to the original, serial, history-based version. The CUDA version has a higher initial overhead and so is less efficient than the serial history-based version at low numbers of Monte Carlo particles. As the number of Monte Carlo particles increases, the CUDA version scales in a super linear fashion. After the number of particles exceeds the batching threshold, the wall clock time of the CUDA version begins scaling linearly as expected.

Figure 6 shows the results of a particle scaling study performed with the optimized Thrust version of the ALPSMC code that was generated from the

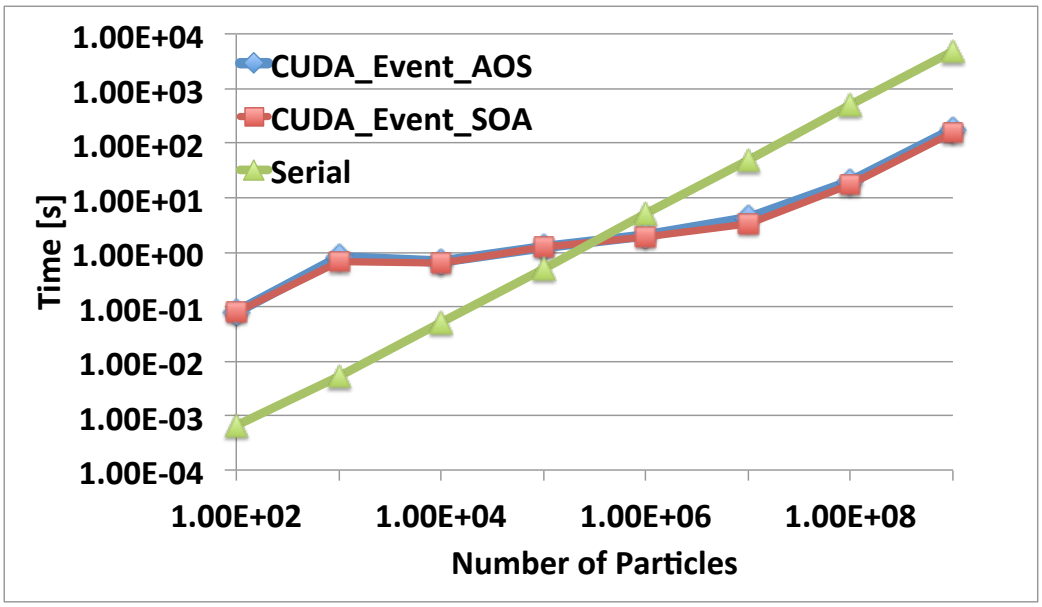


Figure 5. Log-log plot showing wall clock times versus number of particles for the CUDA event-based algorithm using SOA and AOS formats, compared to the original serial history-based algorithm.

optimized CUDA version. In contrast to the results of our previous section, the optimized Thrust version now exhibits the same performance characteristics as the optimized CUDA version. After making these algorithmic transformations, the Thrust version now performed slightly more efficiently than the CUDA version. This outcome demonstrates that an abstraction layer can be performant as long as the code in the abstraction layer uses the same optimizations utilized in the explicit CUDA implementation.

Closely inspecting the results from the explicit CUDA version compared with those from the Thrust CUDA version revealed an interesting and unexpected result. Figure 7 shows the comparison of the CUDA and Thrust CUDA event-based version using the SOA data structures. The Thrust version is slightly faster than the CUDA version for all numbers of particles. We expect that this result is due to two possible factors. First, the Thrust scheduler may be launching kernels more

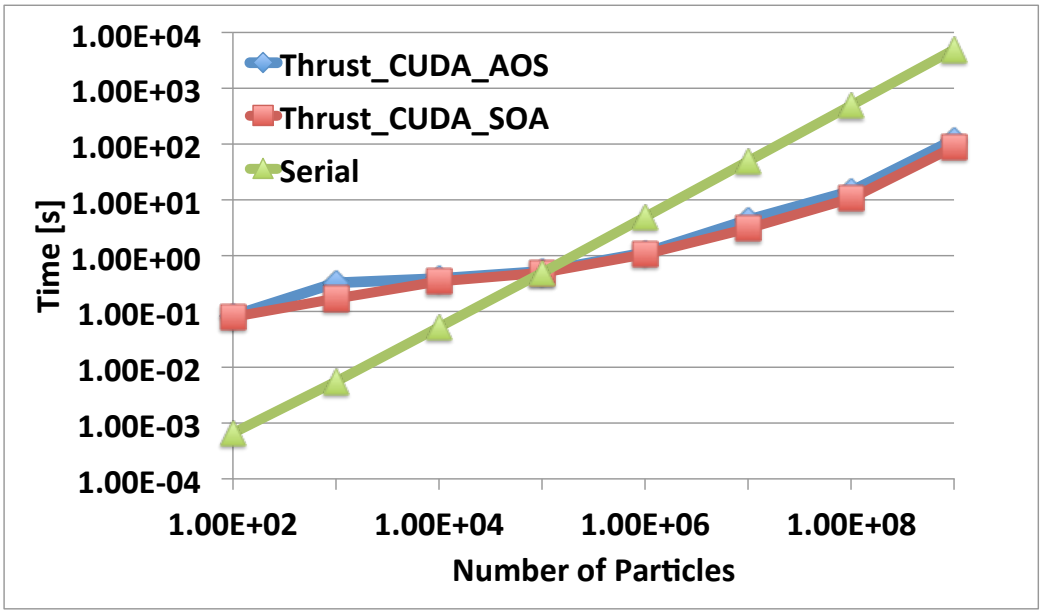


Figure 6. Log-log plot showing wall clock times versus number of particles for the Thrust event-based algorithm running with a CUDA backend using SOA and AOS formats, compared to the serial history-based algorithm.

effectively then the kernel launching scheme we implemented. Second, the memory locations of the read-only tallies and written tallies are stored with the Thrust functor which may allow Thrust to optimize what memory exists in registers or caches when the kernel launches. In the explicit CUDA version, the memory exists in either global memory or the constant memory which is already predetermined.

3.3.3.2 Re-Evaluating the History-Based Method.

Work performed by Nvidia's Anthony Scudiero Scudiero (2016) suggests that it may be possible to achieve performance on GPUs using a history-based Monte Carlo transport algorithm if the correct transformations are made. Additionally, since Monte Carlo transport is a memory-bound problem, using a less compute-optimized approach with lower memory overhead might be a more efficient approach.

To re-evaluate the use of the history-based algorithm on GPUs, we began by making changes suggested by Scudiero (2016). First, we moved those calculations

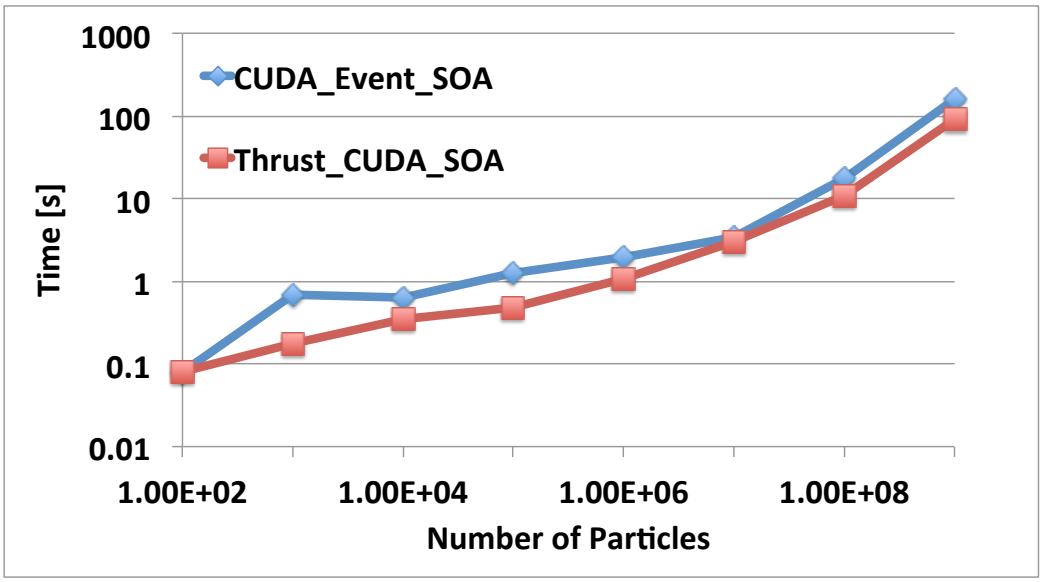


Figure 7. Log-log plot showing wall clock times versus number of particles for the Thrust event-based algorithm running with a CUDA backend compared to the explicit CUDA event-based version both using SOA.

that only needed to be performed once for all particles out of the single large kernel. Second, we utilized shared memory for storing the particle data structure and read-only constant memory for storing the material data (e.g. cross section values). Finally, we removed all atomic tally updates and replaced them with a shared per particle tally that is reduced to single values after the kernel is complete. The results of this work are shown in Figure 8.

The results from this test were very promising. This plot shows that the CUDA history-based algorithm has better scaling with number of particles as well as a significant performance increase at high numbers of particles. At low numbers of particles, the wall time of the CUDA version is constant, regardless of the number of particles, which is due to the overhead involved with accessing the GPU hardware. This overhead is sufficiently low (on the order of 0.01 seconds) that it should only affect problems using an unrealistically low number of particles.

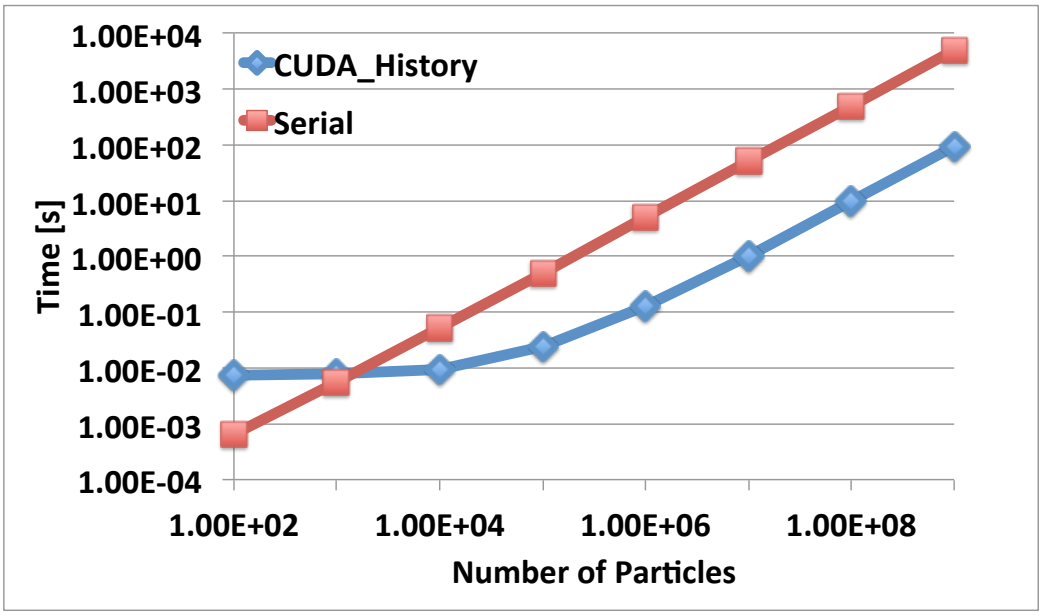


Figure 8. Log-log plot showing wall clock times versus number of particles for the CUDA history-based algorithm compared to the serial history-based algorithm.

3.3.3.3 Thrust and CPU. In order for the Thrust implementation to be considered portable, we must be able to demonstrate the possibility for performance of the Thrust approach on multiple platforms. For this study, we will compared the particle and processor scaling of the Thrust event-based method compiled with both the CUDA backend (for GPUs) and the OpenMP backend (for CPUs) to that of the original serial history-based method running on a single CPU core. The speedups of the event-based method compared to the original serial method are shown in Figure 9. The event-based method is slower on the CPU than the history-based method for a single thread, with a 0.5X speedup corresponding to a 2X slowdown. The Thrust event-based method with two OpenMP threads is roughly equivalent to the original history-based serial method. The Thrust event-based model reaches speedups around 5X at 16 threads when compared to the original serial history-based algorithm.

The speedups of the Thrust event-based method compared to the same Thrust event-based method run serially are shown in Figure 10. The speedups are generally as expected up to four threads but are less than expected at eight and sixteen threads. The Thrust event-based model reaches maximum speedups around 10X compared to the Thrust model run serially.

3.3.4 Results Summary. Table 5 shows the speedups achieved for each method running 10^9 particles on either two Nvidia Tesla K80 GPUs or a CPU with sixteen OpenMP threads. These results demonstrate that a significant amount of performance potential exists in the optimization choices that are made. The use of Thrust as a portability abstraction is not only viable but outperforms the other methods on the GPU. In addition, history-based approaches perform better or at least as well as the event-based approaches on the GPU for this problem. While the event-based Thrust OpenMP results are significantly less than optimal, they do demonstrate portability and some performance gain.

Table 5. Maximum speedups for each approach when compared to the original history-based serial method in ALPSMC

Method	Speedup
CUDA Event SOA	31.32
CUDA History	52.78
Thrust Event CUDA SOA	54.62
Thrust Event OpenMP SOA	5.54

3.4 Conclusions

In this chapter we described our investigations of portable event-based Monte Carlo algorithms implemented using the Nvidia Thrust library in the research Monte Carlo test code, ALPSMC. We found that our initial explicit CUDA implementation of an event-based Monte Carlo algorithm performed significantly more efficiently than a Thrust implementation on GPU platforms,

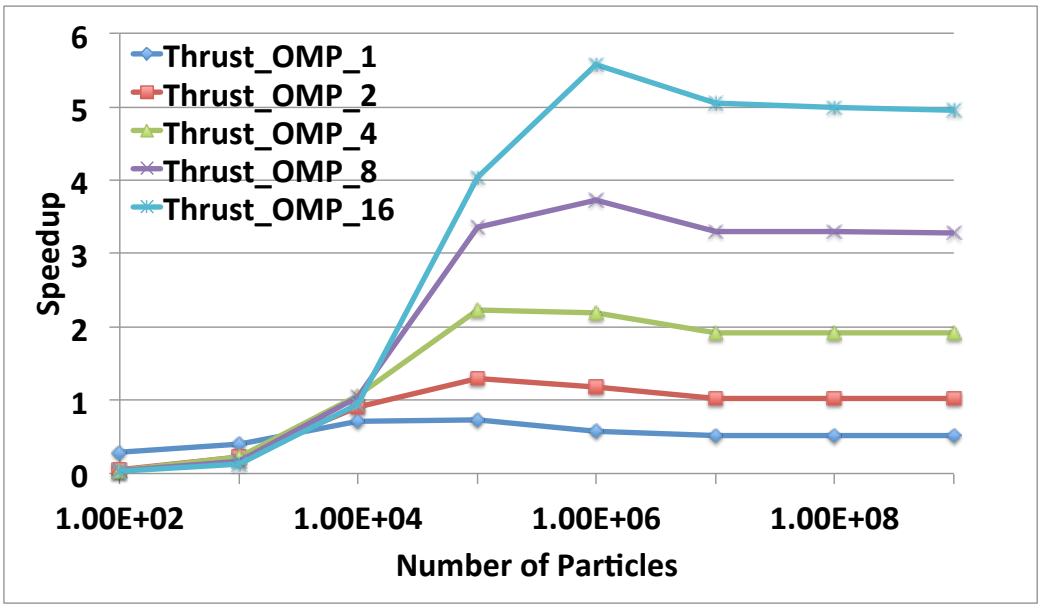


Figure 9. Speedups versus number of particles for the event-based Thrust CPU method with 1, 2, 4, 8, and 16 OpenMP threads compared to the original serial history-based algorithm.

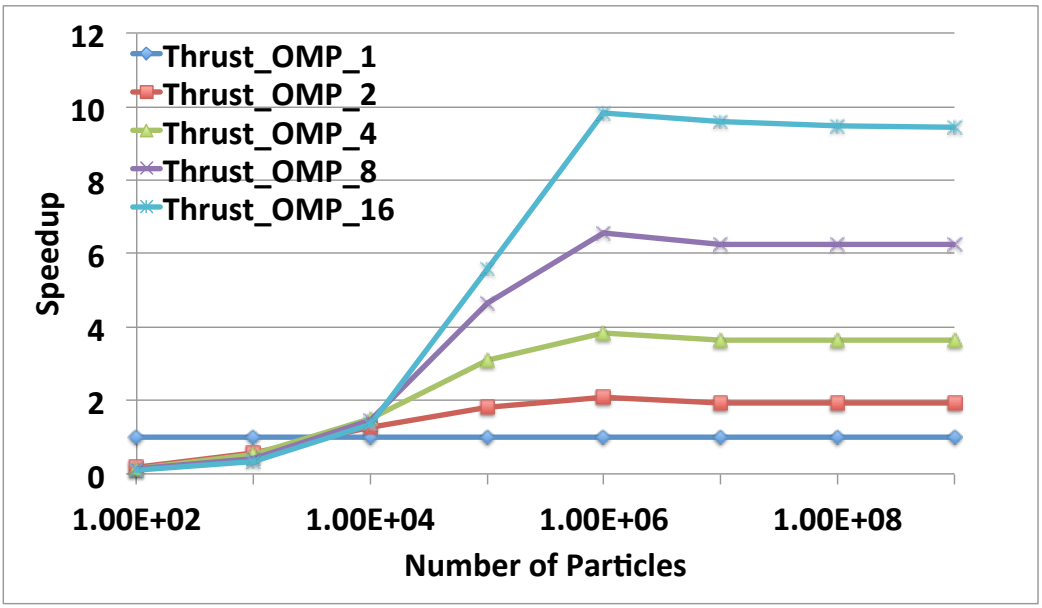


Figure 10. Speedups versus number of particles for the event-based Thrust CPU method with 1, 2, 4, 8, and 16 OpenMP threads compared to the Thrust CPU method serially.

most likely as a result of additional flexibility in access to different memory spaces on the GPU. We have also shown that, with the same optimization choices, the

Thrust abstraction layer can be just as effective as writing native CUDA. This work has shown that the event-based approach for Monte Carlo transport on GPU architectures is viable and can achieve node level speedup results that are acceptable.

While investigating this problem, we also discovered that the performance of the event-based algorithm is affected by what tallies are being used. A zonal scalar flux tally requires atomic operations that significantly impacted the performance of the code, in some cases producing slowdowns instead of speedups. We decided to remove the tally in order to focus on the effectiveness of the event-based algorithm. Chapter V (Data Race Management: Output Tally Data) will perform a deep dive into more effective ways of handling such tallies.

We have also demonstrated that the history-based Monte Carlo transport algorithm can perform efficiently on the GPU. As a result, the history-based approach should be investigated further for use on GPU architectures. For the Monte Carlo test code and numerical problem we investigated, we see even greater speedups with the CUDA history-based approach then we do with the CUDA event-based approach, and it required significantly fewer code modifications.

Finally, we were able to create a portable algorithm that scales with processors on a node level. The event-based approach shows some viability, but does not significantly outpace the history-based approach. Further study in evaluating the history-based approach in a larger application presented first in Chapter IV, and further in Chapter VII, which is important given the possible effects that code size has on this result.

CHAPTER IV

DATA RACE MANAGEMENT: THREADING MODELS

The work presented in this chapter was first published in the International Conference on High Performance Computing & Simulation (HPCS) in July 2019. I was the primary contributor to this work in developing the algorithm, writing the new code, and writing the paper. Dr. Patrick Brantley, Shawn Dawson, Dr. Michael Scott McKinley, Dr. David Richards, and Dr. Matthew O'Brien provided ideas and feedback throughout the development process and assisted in editing the paper. Dr. Hank Childs assisted in editing the paper.

4.1 Introduction

In this chapter we introduce Thin-Threads (defined in Section 4.3.2), a new threading approach for Monte Carlo particle transport problems. While elements of Thin-Threads have appeared in previous research, our contribution lies in combining these elements, providing a thorough description of implementation, and evaluating its efficacy. Additionally, we look at new methods for overlapping computation and communication using Thin-Threads. Finally, we show that the Thin-Threads approach is capable of outperforming the traditional "Fat-Threads" (defined in Section 4.3.1) approach, up to three times faster on CPUs and ten times faster on GPUs for certain workloads.

4.2 Background

The work related to this chapter was conducted in the Quicksilver proxy application, which is described in the article *Co-design at Lawrence Livermore National Lab: Quicksilver* (2017). Quicksilver solves the Monte Carlo particle transport problem by using distributed particle streaming and a multi-group energy nuclear data energy representation. Quicksilver originally implemented threading

through a Fat-Threads model, described in Section 4.3.1. An initial implementation of the Thin-Threads model was added to Quicksilver in order to provide a feasible method for GPU computing. A discussion of the process that led to Thin-Threads as well as the key features of the OpenMP 4.5 and CUDA implementations are presented by Richards et al. (2017).

Quicksilver is a proxy application of the full production code, *Mercury* (2019). Quicksilver was originally developed to model Mercury's call tree and memory usage patterns for streaming multi-group problems. Mercury uses distributed memory particle streaming as well as domain replication to scale across nodes. Additionally, it also uses both continuous energy and multi-group energy cross sections. Mercury implemented threads using OpenMP and the Fat-Threads threading model, described by P. Brantley et al. (2013). Mercury has since implemented the Thin-Threads threading model discussed in this paper.

The importance of this model and primary reason for its development, was to change the unfavorable data access pattern by the previously favored, Fat-Threads model. By designing a threading model around the data access pattern that is more feasible and amenible to the GPU hardware, we are able to better understand and control how data is managed in our Monte Carlo transport applications.

This chapter extends the discussion of this work and clearly defines the elements that make up the Thin-Threads model. By explicitly defining this threading model we can facilitate more discussion on this topic. Additionally, this provides a starting point for others to begin working on Monte Carlo transport problems on GPUs.

4.3 Threading Models

To solve Monte Carlo particle transport problems, millions to billions of particles need to be processed. Parallel computing is necessary to process this number of particles in a reasonable amount of time. Monte Carlo particle transport problems are embarrassingly parallel, since the unit of work — a particle — is completely independent of all others. As supercomputer architectures have shifted to increased parallelism within a node, adding parallelization through threading has become increasingly common and necessary.

There are two major approaches to solving Monte Carlo particle transport problems: history-based and event-based. The work presented in this paper applies the Thin-Threads threading model to the history-based Monte Carlo transport problem. With the history-based tracking algorithm, individual particle histories are tracked until a predetermined amount of particles has been simulated. These particles are processed one at a time, until there are no more particles left to process. For the event-based tracking algorithm, particles are continually regrouped by the event they will process next. With this algorithm, each event group is processed in parallel before needing to regroup particles again.

History-based Monte Carlo particle transport applications generally divide work into three distinct sections: cycle initialize, cycle tracking, and cycle finalize. These three sections are described in pseudocode in Figure 11. Cycle initialize and cycle finalize are both relatively small and straight forward. Cycle initialize handles setting up inputs, such as sourcing particles, and doing variance reduction calculations. Cycle finalize handles reducing output data, such as tallies collected during tracking. Cycle tracking is the core of the code, containing the large majority of the functionality and physics. The work done during cycle tracking is

almost entirely contained within a loop over particles. Inside the loop, each particle computes which event it will do next, via sampling probability distributions and using random numbers to make decisions. Then the particle executes its given event, e.g., moving through the mesh, colliding with the background material, etc. Particles continue to do this two-step process — compute distances then apply the nearest event — until they reach an end condition, such as absorption or census.

```
cycle_init() {
  source in particles
  population control
}

cycle_tracking() {
  for all particles {
    do {
      compute distance to census
      compute distance to facet
      compute distance to reaction
      do segment with shortest distance
      increment tallies
    } until census, absorbed, escaped
  }
}

cycle_finalize() {
  reduce all tallies
}
```

Figure 11. Pseudocode for the three major phases of a history-based Monte Carlo transport algorithm.

Parallelization usually occurs over the "for all particles" loop in cycle_tracking(). Traditionally, particles are split across threads in groups, providing each thread with its own unique chunk of work to complete. We refer to this as the "Fat-Threads" approach, which we describe in more detail in Section 4.3.1. An alternative approach is for threads to share a chunk of particles,

with each thread operating on a single particle within the collection of particles. We refer to this as the Thin-Threads approach, which we describe in Section 4.3.2.

4.3.1 Traditional Fat-Threads Approach.

Overview.

A Fat-Threads threading model is one where all potential data races are handled through replication of data structures. This allows each thread to work completely independently of one another. Each thread is assigned its own collection of particles to work on, and all output tally and buffer type data structures are replicated. Replicating tally data can be non-trivial, as tally data structures exist in multiple forms: tallies for a single value over the whole problem, tallies for each element in the problem, and tallies for each material in the problem. Each of these tallies requires different amounts of memory to store their data. Using this threading model and combining it with a load balancing algorithm, M. J. O'Brien et al. (2013) showed its ability to scale well on CPU platforms, to over 2 million processors.

Quicksilver implements Fat-Threads in a typical fashion. Its fundamental unit of work is advancing a particle, its primary data element is the particle, and its data structure for a particle contains roughly 200 bytes of information. Particles are stored in "particle vaults," which is a container class for grouping particles together and defining functions on sets of particles. At the highest level in the data structure, there is a "particle vault container" (PVC) that can hold a changing number of particle vaults, as well as shared data between vaults. Finally, each rank is given a PVC to organize its workload, and each thread associated with that rank is then given a particle vault from the PVC.

In Monte Carlo transport problems, distributed-memory parallelism is commonly used to split up large geometries into separate domains across ranks. Separate geometric domains add the need for particles to be communicated across ranks as they move through the geometry. In the Fat-Threads model, particles are communicated asynchronously across ranks when needed by the cycle tracking function. When a rank runs out of particle vaults to give to threads, that rank can receive a buffer of particles from another rank, and then fill up new particle vaults, continuing the current cycle. In addition, threads can perform the send and receives themselves as they fill buffers or need more work.

This model for running particles on ranks and threads works well on CPU platforms, by maintaining data locality in a thread and removing the need to deal with data races between threads. The communication cost of sending particles to different ranks is almost completely masked by the computation of particles on each rank, since the computation of particles on each rank occurs while particles are in flight. Additionally, particle vaults become an obvious organization structure for dealing with load balance, providing a flexible infrastructure for running threads.

Barriers on GPUs.

There are two primary concerns with the Fat-Threads model — memory footprint and communication from accelerators.

With respect to memory footprint, the issue is that the Fat-Threads model is likely to use too much memory on GPU devices. When switching from a CPU platform to a GPU platform, the number of threads per rank goes from tens of threads (at most) to thousands of threads or more. If data structures continued to be replicated in the same manner on a GPU platform, providing each GPU thread with its own data structures to read from or write to, then available memory

would quickly run out. This is a concern even if a code extends GPU memory via paging in memory from the host. Even given infinite access to host memory, GPU architectures would struggle from a complete lack of coalesced memory access and a need to constantly page-in data, resulting in an inability to get acceptable performance.

With respect to communication from accelerators, the fundamental issue is the lack of MPI functionality from a GPU device. The GPU cannot make the same MPI function calls that a CPU can during particle tracking. This is particularly problematic for the Fat-Threads model, since it relies heavily on the use of asynchronous communication to move particles from rank to rank while computation is being done. Since the MPI calls cannot be made while processing particles, new methods for communicating particles across boundaries must be investigated.

Between these two issues, the Fat-Threads model appears to be incongruent with GPU architectures.

4.3.2 Thin-Threads.

Overview.

Thin-Threads have multiple beneficial properties for history-based Monte Carlo on GPUs. First, Thin-Threads are threads that are light on memory usage and communication. Second, Thin-Threads handle all potential data races directly, primarily through use of atomics. This model allows for a larger number of threads to be callable at once, reducing the memory footprint when threading. Threads primarily work independently, although there is some interaction via their shared atomic operations. Third, Thin-Threads do not access MPI or other forms of

inter-node communication directly. Instead, Thin-Threads employ a batching and asynchronous communication model.

Overall, Thin-Threads adapt to modern HPC architectures, in that:

- They are lightweight, in order to match decreases in single thread performance.
- Their communication management is aligned with current restrictions (i.e., MPI communication is not possible, or it is possible but not performant).
- Its design accounts for the currently popular use of accelerators, specifically in achieving overlap in communication and computation.

Figure 12 outlines pseudocode for a new cycle tracking function for the Thin-Threads approach.

```
cycle_tracking() {
  while( !done ){
    for each batch {
      Do Kernel
      Do MPI Send
      Do MPI Receive
      Clean Extra Vaults
    }
    test for done
    if( !done )
      Collapse Vaults
  }
}
```

Figure 12. Pseudocode for batching control flow in the Thin-Threads approach. Do Kernel refers to launching the cycle tracking kernel. Clean Extra Vaults refers to the process of ensuring there is adequate space for the next kernel launch. Collapse Vaults refers to the process of reducing the particles in the particle vault container into the minimum number of vaults required to contain them.

While the basic concept of Thin-Threads is relatively straight-forward, it requires significant attention to detail in implementation. The implementation details are described in depth in the following sections.

Basic Implementation Details.

There are two primary tasks required to implement the Thin-Threads model. The first task is to make the tracking loop thread-safe. This requires adding atomics for writing to output tally data and modifying the particle container data structure to allow for threaded reading and writing. The second task is to remove all MPI from within the tracking loop. This requires adding a replacement MPI model after the tracking loop, as well as additional MPI buffers that get filled during the tracking loop. This MPI model is asynchronous and provides the groundwork for a batching model.

Implementation Details - Batching Model.

We built the batch model around three key concepts. First, memory is allocated from the host side, since memory allocations on the GPU are typically slow and limited to device-only memory. Second, the number of particle vaults in the PVC must be capable of being changed dynamically, i.e., we can add particle vaults to a PVC if needed. The number of particles a single rank may see cannot be known in advance and so we must have a flexible system to allow for new particles to be added. Third, we cannot access the MPI region of the code from within the main body of the tracking loop. All MPI must be handled outside the main body of tracking, although we still need a way to handle particles that need to be communicated. Each of these three key concepts are discussed further in the remainder of this section.

In order to satisfy the first key concept, avoiding new data allocation on GPUs, we determined that the number of particles within a given particle vault needs to be fixed. This allows a particle vault to define the group of particles that will execute together in a kernel. A side effect of the fixed vault size is the need for an extra buffer for managing particles created during tracking, since we cannot know the actual number of particles any given cycle will produce. In order to guarantee there is enough space in the extra buffer, we pre-allocate enough particle vaults to handle the case where every particle undergoes the maximum production in a reaction. This can be determined through a heuristic calculation as long as any particle that produces new particles for computation is also added to the new particle list (i.e., its computation is postponed) to guarantee the size of the extra particle list is bounded. The extra vaults and postponing computation of a particle ensures that we will not need to allocate new data during the tracking kernel.

The second key concept, dynamically changing the number of particle vaults in a PVC, must work within the context of the first key concept, data allocations only from the host (i.e., not from the GPUs). To accomplish this, we designed a host-side data structure (the PVC, which is on the host only) that (1) can dynamically change sizes, and (2) always contains enough memory for each kernel on the device (through the particle vaults it contains). More details on specific data structure choices are explained in Section 4.3.2.

In order to satisfy the third key concept, no MPI communication during particle tracking, all MPI was removed from the tracking loop itself. Instead, when a particle leaves a given rank's domain during the tracking loop, it is placed in a buffer. After the tracking loop finishes, the host inspects this buffer and performs the appropriate communication. The size of this buffer has a clear upper bound,

since it cannot exceed the fixed batch size in a single particle vault, i.e., the number of particles needing to be sent via MPI will not exceed the number of particles we are tracking in each batch. In terms of implementation, we create an index list of particles in the kernel which identifies the particles that need to be communicated via MPI, as well as to which neighbor they need to be sent. This simple tuple of data can be generated quickly in the kernel, allowing for faster compute times, at the cost of needing to loop over the index of particles later, on the host, and copying them into MPI buffers. This method has so far not shown itself to be performance critical, spending orders of magnitude less time than the actual kernel compute times.

Implementation Details - Data Structures.

The preceding subsection (4.3.2) defined three key concepts for the Thin-Threads batch model. One of these key concepts enabled growth in the number of particles stored on a given rank. There are two reasons that particle growth on a rank can occur: through reactions in cycle tracking or through receiving particles via MPI communication.

When a new particle is created, it needs to be added into a particle vault. Of course, in our scheme, the GPU cannot allocate new memory. Our solution is to allocate extra particle vaults prior to executing the tracking kernel, and then have the kernel add new particles to these extra particle vaults as it executes. These new particles can then be considered for future processing. Therefore, the particle vault container must not only be dynamic in size, but also must allow direct access for passing in batches to kernels. We use a vector (from the C++ Standard Template Library) of particle vault pointers to handle these requirements. By making a vector of pointers, our particle vault container can change sizes through a two

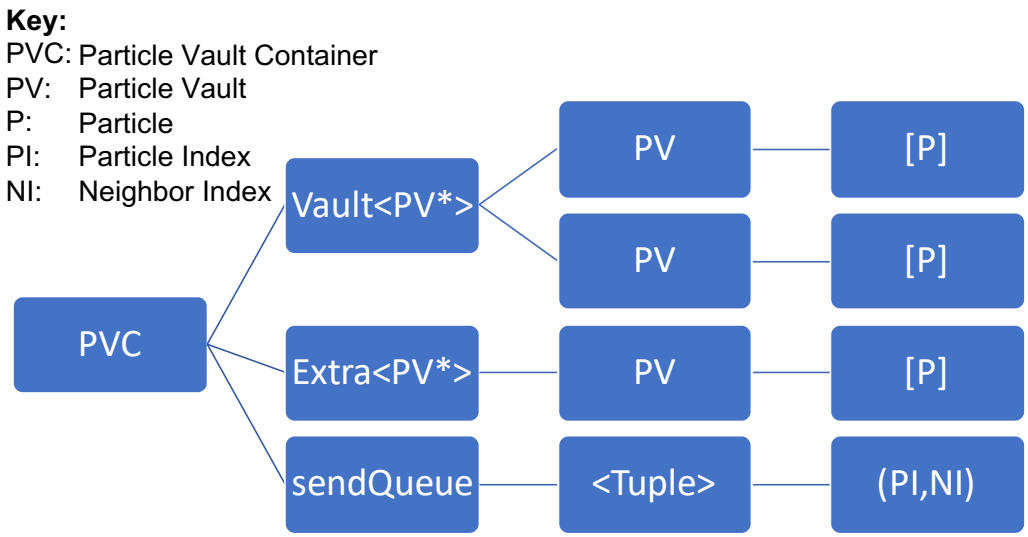


Figure 13. A visual representation of the Particle Vault Container (PVC) data structure.

step process: allocating the pointer (built into the vector class) and then allocating the particle vaults to which the pointers point (custom allocation function). In addition, it allows us to re-organize the vaults in the container as necessary, such as swapping an empty vault for a filled one (which becomes as easy as swapping two pointers instead of needing to perform a deep copy). Figure 13 details the new structure for the particle vault container to enable this new work flow.

After each iteration of kernel launch followed by MPI communication, our algorithm cleans up the extra vaults and combines newly received and newly created particles. This process creates new batches for use in future iterations.

Implementation Details - Control Flow.

Figure 12, which appeared earlier in the Thin-Threads overview section, describes how the Thin-Threads model incorporates batching into the control flow. Its control flow allows for overlapping computation with communication, and lets

us recover lost performance on CPUs, compared to Fat-Threads model. The Thin-Threads control flow works as follows. First, a vault is taken out of the particle vault container and sent into the Kernel (Do Kernel). Second, any particles that need to be sent are pulled into MPI buffers based on the values in the send queue tuples, particles index in vault, and neighbor rank index (Do MPI Send). Third, the host checks to see whether or not any particles need to be received (Do MPI Receive). Fourth, newly created particles and received particles are condensed into particle vaults that are then added to the PVC. The extra particle vaults are populated again with all empty vaults (Clean Extra Vaults). Once this process has been completed, the data structures are ready to handle another pass through this process, i.e., a filled vault is ready for kernel launches, and extra vaults are ready to receive new particles.

One significant element of this application is the need to run on the CPUs and GPUs through a single source code base. To achieve this, a simple execution policy model was established which allowed for ranks to determine what form the kernel would take. The amount of replicated code for each policy available was reduced to a single function call inside each kernel or for loop. This means that each policy only needs to define the parameters necessary to launch the kernel, or run the for loop. The available policies are Serial, OpenMP 2.0, OpenMP 4.5, and CUDA. The use of macros around language specific functions, such as atomics, allows each of these methods to run through the same code on CPUs as well as GPUs. Additionally, this execution policy model will be the basis of future work, where we can explore the use of CPUs and GPUs at the same time.

4.4 Thin-Threads Performance Studies

This section describes the results from studies performed on Lawrence Livermore's IBM/Nvidia GPU test platform, Ray. This machine uses two IBM Power8 CPUs and four Nvidia Pascal P100 GPUs per node. The IBM Power8 CPU has 10 cores and can run up to 8 threads per core. That said, our best performance comes from running threaded CPU runs with four threads per core, and so we only use four of the eight threads in our experiments.

A Monte Carlo particle transport workload is defined by two major factors — the types of reactions and likelihood of mesh facet crossings. For the types of reactions, the key elements are the cross section and material information. For the likelihood of mesh-facet crossings, the key elements are the mesh layout and decomposition. While the elements defining the types of reaction are defined by the underlying physics, the elements defining the likelihood of mesh-facet crossings can be varied. Therefore, our performance study varies the elements behind mesh-facet crossings (mesh layout and decomposition).

For the material and cross section information, we considered the Godiva in water problem as defined by Cullen, Clouse, Procassini, and Little (2003). Specifically, we replicated the ratios of particle streaming to collisions, as well as the ratios of the types of reactions that occur in the collisions.

For the mesh-facet crossings, we defined the size of the mesh elements so that the likelihood of events is roughly equal (i.e., so the occurrences of mesh facet crossing and collision events are balanced). The problem defines a Cartesian mesh of 10x10x10 mesh elements per rank (one decomposition element) in a rectangular, doubling, scaling pattern. For example, one rank would use [10x10x10] mesh elements, where as two ranks would use [20x10x10] mesh elements, and four ranks

would use [20x20x10] elements. Given the simplicity of running problems in a rank per GPU mode we opted to use four ranks for the base problem and define one node worth of performance as the result of running on four P100 Pascal GPUs. In order to maintain a fair comparison when running on CPUs, we opted to also use four ranks per node and use OpenMP threading to fully utilize a node. At four threads per core and five cores per rank, the CPU data was generated using four ranks with twenty threads per rank.

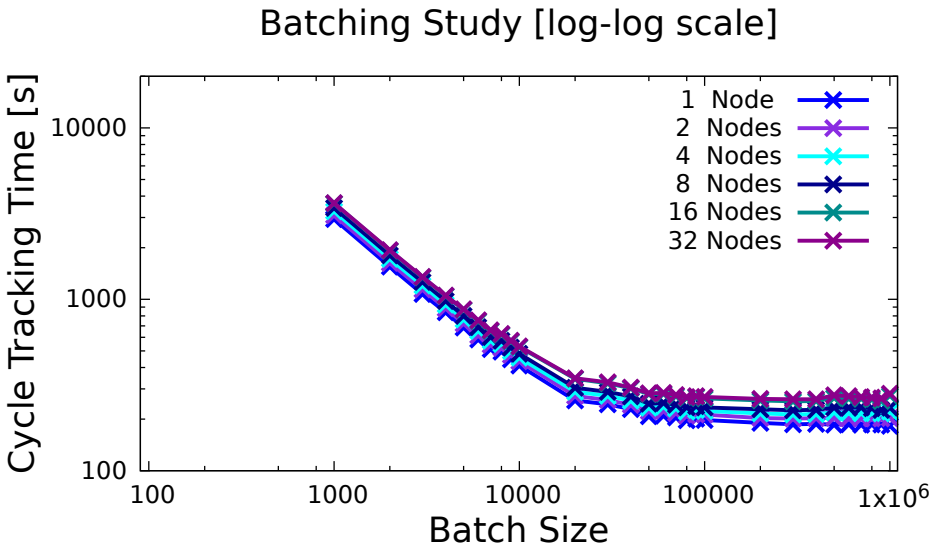
In terms of runtime per cycle, our goal was to pick workloads that reflected real world problems. On the one hand, runtimes that are very short would not reflect real world problems (and also skew analysis). On the other, long runtimes, while more common in practice, limit the number of tests we could perform. Overall, we decided to consider runtimes of approximately two seconds per cycle. To accomplish this, we opted to run one million particles per rank, which completes in roughly two seconds per cycle on a GPU. Given four ranks per node as our baseline, we ran four million particles per node and scale accordingly during scaling studies.

4.4.1 Effect of Batch Size on Performance. In this section, we analyze the effect batch size has on the overall performance for Thin-Threads. Batch size has multiple, potential impacts on performance. First, it determines the number of threads that can be running simultaneously on a rank, which has a profound impact on the performance of threading. Second, it allows for different amounts of computation to overlap with communication, providing a tunable knob for optimizing MPI. Finally, batch size choices also determine the number and size of memory allocations that need to occur, which should be minimized in this setting. The results for this section are plotted in Figures 14a and 14b. In

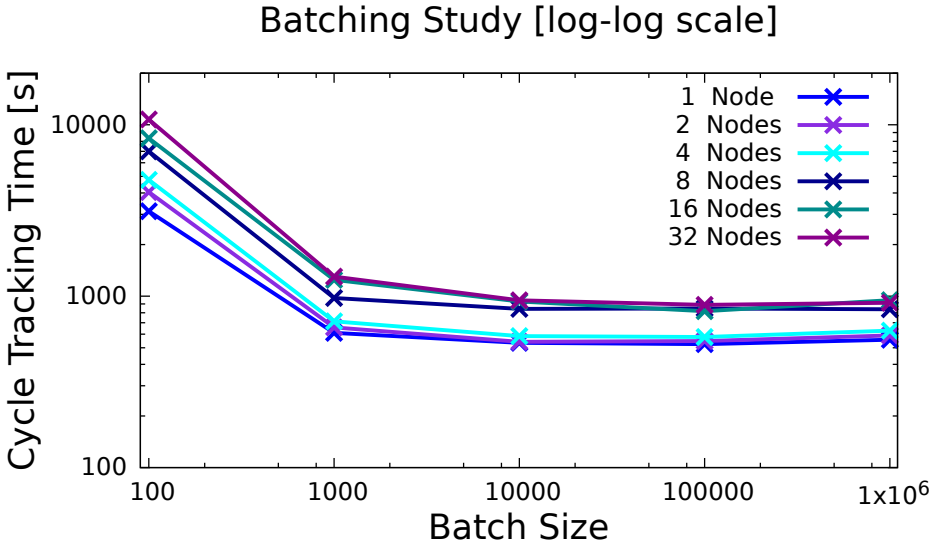
these figures, batch size is plotted on the x-axis and runtime in seconds on the y-axis; with respect to performance, lower is better. The experiments performed were somewhat asymmetric: our minimum batch size was 100 for the CPU and 1000 for the GPU. We had to increase the minimum batch size for the GPU, since batch sizes of 100 did not complete within a reasonable amount of time, due to not utilizing the GPU adequately.

The effect of batch size on the availability of threads has profound performance implications. This is especially true on GPU architectures, as the batch size determines the kernel size of the particle vaults. Large kernels are needed to efficiently utilize all of the cores on GPU hardware. Figure 14a clearly shows the trend of increased performance (decreased runtime) as the batch size increases. The trend in runtime decreases linearly as we increase batch size, up until the GPU hardware is adequately saturated. Once GPU has enough work (at around a batch size of 50,000), the performance benefit plateaus. Batch sizes above 50,000 provide similar performance, reducing the need to find a specific value for optimum performance. At higher batch sizes (approaching one million), the curve trends up slightly, most likely due to there being less MPI overlap occurring in that regime.

Figure 14b shows the performance trends on CPU architectures at different scales. The trends for CPUs have a similar shape to GPUs. The primary difference between the two architectures is that the maximum performance (lowest runtime) point for CPUs occurs much earlier than it does for GPUs. Both sets of results show a decrease in performance (increase in runtime) as when batch sizes become much smaller than the total number of particles. For CPUs, our results consider batch sizes as small as 100. Increasing the batch size to 1000 results in almost an order of magnitude increase in performance.



(a) GPU



(b) CPU

Figure 14. These plots show weak scaling studies of cycle tracking time versus batch size, for 1 to 32 Nodes. This data shows batch size has considerable impact on performance. For GPU runs (sub-figure (a)), the optimum batch size is 300,000 particles per batch. For CPU runs (sub-figure (b)), 100,000 particles per batch was the optimum size, although the performance differences for batch sizes over 1000 were much smaller. The most important takeaway from these plots are the trends across all nodes, rather than the line corresponding to a single configuration of nodes.

Another interesting point is that this trend shows nearly identical performance at different scales, meaning that even at poor batch sizes for GPU performance the MPI weak scaling is still managing well. This is true on CPUs as well when running 4 ranks per node. In our previous experience, not shown here, we witnessed negative side effects to running with batch sizes that were too large when run at large scale (thousands of ranks).

Table 6. Weak scaling results for Thin-Threads on CPUs and GPUs, compared to the weak scaling results for Fat-Threads for the same configuration. Time is listed in seconds. A batch size of 300,000 was used for the Thin-Thread+GPU runs and 100,000 was used for the Thin-Thread+CPU runs. There was no batching in the Fat-Thread code. All models were run with four ranks, and the CPU runs used 20 threads per rank.

Nodes / Ranks	Thin (GPU) [s]	Thin (CPU) [s]	Fat (CPU) [s]
1 / 4	1.866e+02	5.247e+02	6.788e+02
2 / 8	2.013e+02	5.470e+02	8.531e+02
4 / 16	2.130e+02	5.777e+02	1.998e+03
8 / 32	2.250e+02	8.482e+02	2.380e+03
16 / 64	2.537e+02	8.166e+02	2.327e+03
32 / 128	2.610e+02	8.902e+02	2.725e+03

4.4.2 Weak Scaling Efficiency Comparisons. In this next phase of results, we consider two topics: weak scaling and comparison to Fat-Threads. Our experiments here incorporated the optimum batch sizes from the previous phase of results (Section 4.4.1).

Table 6 lists the results from a weak scaling study (1 node to 32), comparing the same configuration for Thin-Threads with GPUs, Thin-Threads with CPUs, and Fat-Threads on CPUs. The entries in each table are the actual runtimes. This table highlights the added benefits of the Thin-Threads model, especially at this

Table 7. Efficiency data from the weak scaling study. Basic parallel efficiency is given by comparing to single node performance. Relative efficiency is given by comparing to previous size performance (i.e., 2 nodes efficiency is tracking time as [1 Node / 2 Nodes], whereas 4 node efficiency is tracking time as [2 Nodes / 4 Nodes]).

Nodes Ranks	Thin (GPU) Eff. 1 Node	Thin (GPU) Rel. Eff.	Thin (CPU) Eff. 1 Node	Thin (CPU) Rel. Eff.	Fat (CPU) Eff. 1 Node	Fat (CPU) Rel. Eff.
1 / 4	100%	—	100%	—	100%	—
2 / 8	92.69%	92.69%	95.92%	95.92%	79.56%	79.56%
4 / 16	87.61%	94.51%	90.83%	94.69%	33.97%	42.70%
8 / 32	82.93%	94.67%	61.86%	68.11%	28.52%	83.95%
16 / 64	73.55%	88.69%	64.25%	103.9%	29.17%	102.3%
32 / 128	71.49%	97.20%	58.94%	91.73%	24.92%	85.39%

scale, as even the CPU results show improvement over the original Fat-Threads model.

The data is most representative of real-world workloads at higher node counts. With low node counts, each node's domain has fewer neighbors, which means less time is spent doing communication. For example, with four nodes, each node's domain has only two neighbors. As the node counts get higher and higher, then most of the nodes will have six neighbors (+/-X, +/-Y, +/-Z). In particular, slowdowns in performance can be seen at 8 and 16 nodes, as nodes at these levels of concurrency have more neighbors than smaller concurrencies. Specifically, at 16 nodes and 64 ranks has ranks that needs to send and receive messages with up to

six neighbors. We can see this effect on the weak scaling data, especially for the Thin-Threaded CPU code, as the complexity of the MPI increases the runtime increases to match but settles again at a new steady value.

Table 6 shows that, for 32 nodes and 128 ranks, the GPUs running Thin-Threads are 3.4× faster than the CPUs running Thin-Threads and 10.4× faster than this same configuration of CPUs running Fat-Threads. We consider this performance to be very successful in the context of Monte Carlo particle transport. Since the Monte Carlo particle transport algorithm is not bound by the resources that the GPU makes readily available (compute and streaming memory throughput), it is inherently difficult to achieve significant GPU performance. Instead, it is bound by memory latency and filled with branching divergent paths, both of which are identifiable as significant limiting factors with this algorithm on GPUs.

Table 7 shows the scaling efficiency up to 32 nodes. This can be calculated directly from Table 6. The efficiency is calculated using a single node as a baseline. This table shows that the GPU maintains a weak scaling value of just over 70% efficiency at 32 nodes compared to using just one node. On a CPU platform, this is just under 60% for Thin-Threads and only 25% for Fat-Threads. This drop in performance on CPU platforms is in part due to the greater sensitivity that the CPU performance is showing to the added MPI complexity of higher scales, as well as the fact that 4 ranks, with 20 threads per rank, is not the optimum CPU layout for this machine.

Table 7 also shows the relative efficiency of scaling, for each increase in node count. This table highlights a number of interesting points about the scaling pattern. That said, some of the effects are due to the relationship between node

count and problem size. As we increase the problem by a factor of 2, we are doing so only in one dimension at a time. At 16 nodes we have a perfect cube for problem dimensions [4x4x4], where as at 8 or 32 nodes we have a rectangular problem domain instead ([4x4x2] and [8x4x4], respectively). This informs some of the findings of the table. First, not all of the scales are slower. Specifically, on the CPU runs, the 16 nodes experiments shows better performance than the 8 or 32 node runs. This is most likely a side effect of load balancing in the scaling study itself. Second, there are a few definite points where efficiency drops dramatically compared to the previous scale. This highlights a step in complexity, as the subsequent scales do not continue to drop dramatically.

An important take away from this efficiency data is that the Thin-Threaded model exhibits promising scaling behavior. This data shows the viability of this approach and that under these circumstances Thin-Threads performs best. That said, Monte Carlo particle transport problems can have irregular performance behaviors, and a more comprehensive study at higher node counts and more workloads could be useful.

4.4.3 Weak Scaling on BGQ. This section describes results on Lawrence Livermore's Vulcan machine, which uses the BGQ architecture. Table 8 shows the scaling data we gathered. The performance data comes from a similar workload as was run in Section 4.4.2, with the only significant difference being less particles per node. This change was necessary since a node of BGQ is less performant than a GPU node on Ray.

Our data in this section is presented with respect to a figure of merit (FOM), specifically how many segments per second each problem ran on average. One advantage to considering results with respect to the FOM is that a doubling

Table 8. Figure of Merit and efficiency data from weak scaling runs on Vulcan, with 4 ranks per node and 16 threads per rank. Efficiency against the 1 node runs and the relative efficiency for each step are shown. Relative efficiency is calculated in the same way as described in Table 7.

Nodes	FOM [seg/sec]	Eff. 1 Node	Rel. Eff.
1	2.068e+06	100%	–
2	4.018e+06	97.15%	97.15%
4	7.622e+06	92.14%	94.85%
8	1.443e+07	87.22%	94.66%
16	2.773e+07	83.81%	96.08%
32	5.447e+07	82.31%	98.21%
64	1.072e+08	81.00%	98.40%
128	2.124e+08	80.24%	99.07%
256	4.216e+08	79.64%	99.25%
512	8.359e+08	78.95%	99.13%
1024	1.665e+09	78.63%	99.59%
2048	3.314e+09	78.25%	99.52%
4096	6.600e+09	77.92%	99.58%
8192	1.301e+10	76.80%	98.56%
16384	2.612e+10	77.09%	100.38%
24576	3.909e+10	76.91%	99.77%

in resources should produce a doubling in the FOM. This is represented as percent efficiency — 100% efficiency means double the nodes led to a doubling of the FOM.

This data shows that the Thin-Threads solution scales well up to the entirety of the Vulcan portion of the Sequoia supercomputer. While we see higher efficiency on Vulcan than on Ray, this is most likely due to the nature of each machine. The BGQ system is designed from the ground up to minimize per-node variation in performance and has advanced networking features allowing codes to scale efficiently. Ray does not have these advantages — it has variation in performance per node (since it has power-based CPU clock throttling) and it has a simpler network architecture. Since Ray is a test bed machine, it is likely that

some of our efficiency loss comes from the unoptimized network setup, or clock speed throttling resulting from using more and more of the system. Despite these differences, we see similar performance patterns between the two systems.

Comparing the CPU Thin-Threaded results on Ray with the CPU results on Vulcan, we can see that the same pattern of decreased efficiency at low scales with a leveling out of performance as we increase the scale. Given the similarity in these data sets we believe that a larger system could expect similar scaling performance even at much higher scales.

4.5 Conclusion

In this chapter we demonstrated the effectiveness of the Thin-Threads approach for history-based Monte Carlo particle transport problems on GPUs. Additionally, Thin-Threads have also shown a degree of portability as both CPU and GPU forms of this approach have proved to be performant. On GPU platforms we achieved about 3× greater performance over the Thin-Threads CPU model and about 10× greater performance over the Fat-Threads CPU model.

One reason the Thin-Threads approach was effective was the inclusion of an asynchronous MPI batching model. The batching scheme presented in this paper has the added benefit of being a tunable parameter. This means that for problems where MPI is a dominating factor for performance, finding a good batch size could provide a starting point for optimizing performance. In some cases, we also noticed that the batch size was not a significant factor in performance. In these cases, as long as the batch size provided adequate parallelism, other factors dominated performance aside from time spent in MPI, therefore, overlapping computation with communication had little effect. Even in these cases, however, providing enough parallelism is an important factor and so it is important to determine a good batch

size for the hardware. Through our experiences on Ray and Vulcan we saw that batch sizes of 100,000 or more worked well for GPU platforms and batch sizes greater than one thousand worked well for CPU platforms.

An important aspect of our study on Thin-Thread performance was the parallel efficiency when scaling up to large numbers of nodes. Specifically, we wanted to evaluate the performance of our new Batching+Asynchronous MPI approach. On Vulcan we showed that we could maintain nearly perfect relative efficiency (most being at or greater than 99%) and an overall parallel efficiency of greater than 75% on 24 thousand nodes (98304 ranks) when compared to a single node. On Ray we found we could maintain relative efficiencies in the 90% range after the initial dip around 8 nodes, and maintained a greater than 70% efficiency at 32 nodes on GPUs.

The performance and scalable efficiency of the Thin-Threads approach provides the basis to move forward in developing a GPU version of the full production application, using a single code base and threading model for both the CPU and GPU. The Thin-Threads model was developed inside of the Quicksilver mini-app available on Github (See: *Quicksilver. A proxy app for the Monte Carlo Transport Code, Mercury. LLNL-CODE-684037* (2017)).

CHAPTER V

DATA RACE MANAGEMENT: OUTPUT TALLY DATA

The work presented in this chapter is from an unpublished work. I was the primary contributor to this work in developing the tests, writing the new code, and writing the paper. Dr. Patrick Brantley, Dr. David Richards, and Dr. Matthew O'Brien provided ideas and feedback throughout the development process. Dr. Hank Childs assisted in editing this work.

5.1 Introduction

This chapter presents a new method for managing tally output data, called variable replication. This method has the ability to mitigate performance concerns by providing a new parameter that can be used to trade memory for performance. This ability is crucial to the performance of Monte Carlo transport when the cost of collecting tallies becomes high. The remainder of this chapter is organized as follows. section 5.2 (Motivation) explains the need for this work. section 5.3 (Variable Replication) describes the variable replication method itself. Finally, section 5.4 (Understanding Atomic Performance) describes a study on the behavior of atomics, to better understand the conditions where variable replication is needed and useful.

5.2 Motivation

For the Fat-Threads model, described in subsection 4.3.1 (Traditional Fat-Threads Approach), tally data is fully replicated. This means that each thread has its own copy of all of the tally data. Further since tallies are distributed across multiple resources, reductions are required to get the final result. The introduction of the Thin-Threads model, see subsection 4.3.2 (Thin-Threads), moved away from full replication. That said, moving to a heavy reliance on atomic operations

brought with it significant performance concerns. In particular atomic operations on double-precision data was detrimental to an application's performance prior to NVIDIA's Pascal architecture. This was mainly due to lack of support for double precision atomics in the hardware, leaving only software implementations which were incredibly costly to use. With the inclusion of double precision support in the Pascal generation hardware and above, our understanding about the effect of atomic performance needs to be revisited.

5.3 Variable Replication

Variable replication is the idea that tally data can be replicated less than the total number of threads. Access to tally memory still requires atomic operations, but as the number of replications increases, the likelihood of any two threads contesting that memory decreases. For example, if N threads all write to a single memory location then there is at most N collisions occurring on that atomic write, essentially serializing that operation. If instead of a single memory location, all of the threads write to two separate locations, we have effectively reduced the number of possible conflicts to $N/2$.

This concept is a simple extension of the tally data, that provides a user settable parameter to determine the amount of times any given tally is replicated. The idea is that by increasing this value, the total memory usage will increase, but so too will the performance be improved. This provides a good way to trade some memory for performance, without requiring full replication of the data for every thread of execution. Additionally, it ensures that we can still operate in a Thin-Threads threading model by relying on atomics for data management.

In our implementation, tallies are defined by types and there is a parameter for each type to determine the number of times that tally will be replicated. Also,

access to tally memory is based on a threads local thread_id. In order to access a tally, a thread uses an index value that is its thread_id modulo the number of times that tally was replicated. In this way, threads that are near each other, such as in a GPU warp, are going to request memory that is not the same as its neighbors given a large enough number of replications available.

5.4 Understanding Atomic Performance

To test atomic performance, we ran trials in Quicksilver using multiple levels of variable replication as well as with and without atomics enabled. We disabled atomic operations by encapsulating them in a MACRO and compiling them out. This had the side effect of producing the wrong answers in the output tally data, but did not effect the execution path of the code. On Intel and IBM CPU platforms we found no performance difference when running Quicksilver with variable replications — Balance Tallies 16x replicated and Mesh based tallies 1x replicated — compared to running with atomics compiled out completely. Similarly, on the Nvidia Pascal GPU, we found that the impact of atomics on performance not measurable. From this study we can conclude that atomics will not significantly and negatively impact performance of Monte Carlo transport problems, and therefore are a useful method for handling output tally data.

To verify our current conclusion, that atomics with variable replication will work well for handling output tallies in Quicksilver, we created seven small isolated kernels to test atomics under a well known set of circumstances. Through a combination of seven kernels, we test the impact of data size, coalesced reads, and multiple divergent reads while performing an atomic reduction. The specific aspects that each kernel represents are described in Table 9. The seven reduction

kernels can additionally be described by how they acquire the data for the atomic reduction:

Simple: A hardcoded number

Array 1: Read from an array of doubles, coalesced

Array 2: Read from an array of doubles, random

Class 1: Read from a large class, coalesced

Class 2: Read from a large class, random

Search 1: Linear search on array of doubles

Search 2: Linear search on array of large objects

Table 9. This table describes the key features that each kernel highlights. Data size small = 1 Double. Data size large = 26 Doubles + 26 Ints. For the case of the Linear search kernels each read is of a consistent size but happens a random number of times. Additionally, as data is read linearly in the search coalescing is possible but caching is more likely to help as early on each thread will need to read the same data.

Kernel	Size of Data	Coalesced Read	Divergence
Simple	Small	N.A.	None
Array 1	Small	Yes	None
Array 2	Small	No	None
Class 1	Large	Yes	None
Class 2	Large	No	None
Search 1	Many Small	N.A.	Yes
Search 2	Many Large	N.A.	Yes

In order to measure the impact that atomic contention has on the overall performance, the variable replication scheme was added to each kernel. In this

scheme, the atomic reduction is done into an array where the index being written to is given by the number of replications modulo the threads index. In this way, as more replications are added the overall atomic contention is reduced as well as the contention in each WARP of threads. The data in Table 10 summarizes the findings after running an ensemble of runs for each kernel at replication levels from 1 to 1024 by powers of 2. Additionally, each kernel was run with its own unique optimum choice for deciding thread-block layouts for best performance of that kernel, which was found in advance.

Table 10. This table is a summary of findings from the ensemble runs used to understand the impact of each kernel.

Kernel	Measurable Impact
Simple	Significant
Array 1	Significant
Array 2	Minor
Class 1	Unmeasurable
Class 2	Unmeasurable
Search 1	Unmeasurable
Search 2	Unmeasurable

Table 10 and Figures 15, 16, and 17 provide concrete evidence for the usability of atomics in Quicksilver. Of the seven kernels tested in this study, only the kernels with little to no memory latency issues showed any adverse effects due to atomic contention. Every kernel with larger or multiple memory reads was completely unaffected. Additionally, the edge case of a single double being read from coalesced memory versus from random memory addresses shows that even a small amount of additional memory latency almost completely mitigates any performance loss due to atomic contention.

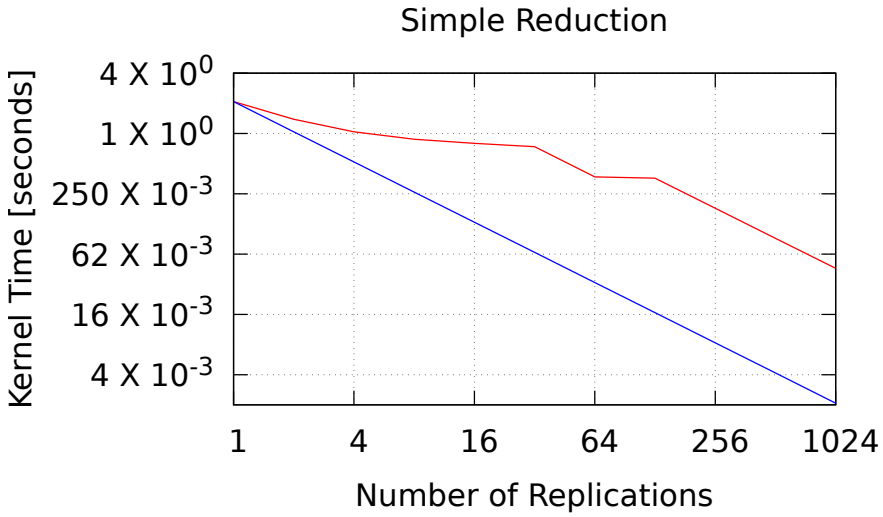


Figure 15. Log-Log plot of total kernel time versus number of replications. 2× number of replications halves the atomic contention. The slope of this curve closely following this 2× drop in runtime, meaning that atomic contention is dominating the performance for this kernel. The Simple and Array 1 kernels both show this behavior, and benefit from ways to mitigate contention, such as replication.

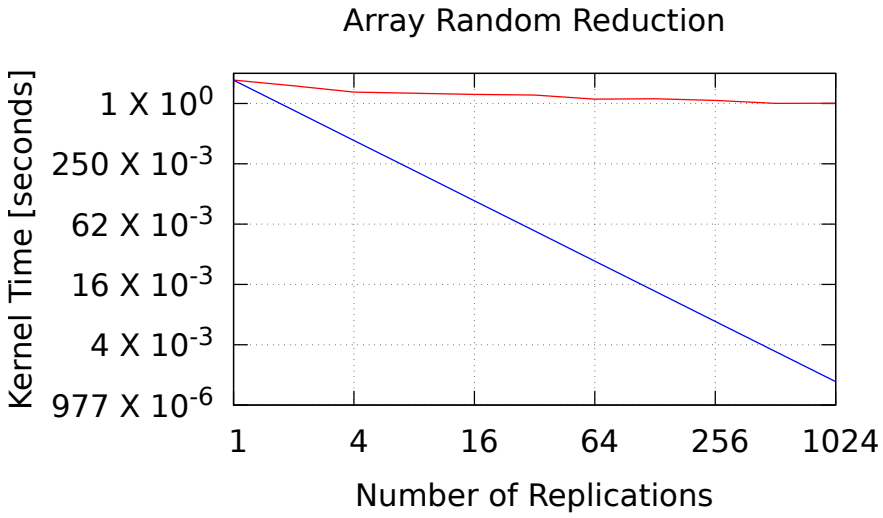


Figure 16. Log-Log plot of total kernel time versus number of replications. 2× number of replications does not reduces the atomic contention by a factor of 2. Instead, we see only a very small benefit from adding replication. This indicates that atomic contention is only mildly affecting overall kernel performance and that other factors affect performance more. The Array 2 kernel shows this behavior.

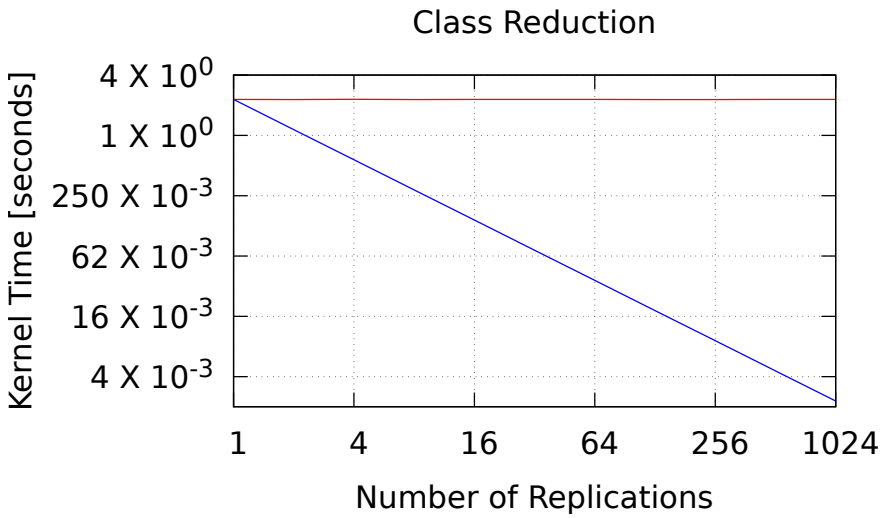


Figure 17. Log-Log plot of total kernel time versus number of replications. 2× number of replications does not affect performance at all. This indicates that atomic contention is either not an issue, or so small of an effect that it is dwarfed by the other aspects of the kernel. The kernels that have this behavior characteristic are: Class 1, Class 2, Search 1, and Search 2.

Monte Carlo transport codes are filled with many memory reads of small and large sizes from random locations in memory. Due to the nature of the algorithm there are not any good ways to mitigate this memory latency issue. Given this, some amount of atomic usage in the kernel should not negatively affect the overall performance. With out study, we demonstrated that variable replication is a useful tool for mitigating the overall impact of atomic, for the cases where they do impact performance. By extension then atomics are a very useful way of dealing with the race conditions associated with writing to output tallies, and variable replication can be used to mitigate any performance losses seen using this strategy.

CHAPTER VI

HETEROGENEOUS ARCHITECTURE UTILIZATION

The work presented in this chapter is from a paper in-submission to the International Conference on Computational Science (ICCS) in June 2021. I was the primary contributor to this work in developing the algorithm, writing the new code, and writing the paper. Dr. Patrick Brantley, Dr. Matthew O'Brien, and Dr. Hank Childs provided ideas and feedback throughout the development process and assisted in editing the paper.

6.1 Introduction

In this chapter we will present a new load balancing algorithm to tackle the problem of dynamic replication in a domain decomposed, heterogeneous environment. In section 6.2 (Motivation), we outline the reason behind choosing to use all of a heterogeneous node architecture for computation, and show a recent work that started the process of making this possible. In section 6.3 (Background), we provide background information about the use of domain decomposition algorithms in Monte Carlo transport. Then in section 6.4 (Related Works), we provide the necessary background information to understand where this work fits in based on previously existing algorithms and developments. In section 6.5 (Our Method), we describe our contributions to the three steps that make up a dynamic replication approach. More specifically, in subsection 6.5.1 (Step 1: Assignment) we describe our new load balancing algorithm, in subsection 6.5.2 (Step 2: Distribution) the single domain load balancing algorithm is described, and in subsection 6.5.3 (Step 3: Mapping) we describe our new mapping algorithm. Finally, the details that describe our experiments are explained in section 6.6

(Experiment Overview), with the result of using this algorithm in practice presented in section 6.7 (Results).

6.2 Motivation

A typical strategy for a heterogenous supercomputer is to use the CPUs only for management and communication with other compute nodes and to use the GPUs to transport particles. This approach usually pairs each GPU with one CPU core to drive the application, and leaves the rest of the CPU cores idle. Based on the relative FLOPS, utilizing the CPU only for management tasks would appear to be an acceptable strategy. Using *Livermore Computing Center High Performance Computing: RZAnsel* (2020) supercomputer as an example, the GPUs make up 1,512 TFLOPS, while the CPUs make up 58 TFLOPS, for a total system GPU+CPU count of 1,570 TFLOPS. This means that GPUs and CPUs make up 96.3% and 3.7% of the total FLOPS, respectively — the programmer effort to engage the CPU may not be viewed as worthwhile. However, CPUs have other benefits, including increased memory size and reduced latency to access memory. Further, many operations for Monte Carlo photon transport are not FLOP-bound. In all, engaging CPUs to carry out computation has the potential to add benefits beyond their FLOPS contributions (e.g., beyond 3.7%).

M. O'Brien et al. (2019) were the first to demonstrate benefits from incorporating CPUs alongside GPUs to carry out Monte Carlo photon transport. That said, their algorithm was limited in utility, because it could only be applied to meshes that could fit entirely within GPU memory. This limitation is crucial in the context of supercomputers, since typical simulations at large scale use computational meshes that exceed GPU memory. Such meshes are decomposed into domains (or blocks), with each block small enough to fit within memory and

each compute node working on one (or more) blocks. This domain decomposition complicates execution, as each compute node can only transport particles where it has valid data. In this paper, we expand upon the work by O'Brien et al. to deal with domain-decomposed meshes. We accomplish this by introducing two new algorithms: one for load balancing and one for building communication graphs. We also analyze the effects of domain decomposition on the performance of hybrid heterogeneous approaches. In all, the contribution of this work is a practical algorithm that translates the potential demonstrated by O'Brien et al. into a real world setting.

6.3 Background

Monte Carlo photon transport problems divide their spatial domains amongst its compute resources (i.e., MPI Ranks) in a non-traditional manner. In many physics simulations, there is a one-to-one mapping between compute resources and spatial domains — a physics simulation with N compute resources has N spatial domains, and each compute resource has its own unique spatial domain. With Monte Carlo photon transport problems, the full mesh is often too large to fit into one compute resource's memory, but not so large that it must be fully partitioned across the total memory of all the compute nodes. Saying it another way, there are often fewer spatial domains than compute resources, and so multiple computational resources can operate on the same domain at the same time. Consider a simple example with two spatial domains (D_0 and D_1) and four compute resources (P_0, P_1, P_2, and P_3). One possible assignment is for D_0 to be on P_0, P_1, and P_2 and D_1 to be on P_3, another possible assignment is for D_0 to be P_0 and P_1 and D_1 to be on P_2 and P_3, and so on. Overall, domain assignment is an additional component for optimizing performance.

In the Monte Carlo community, the mapping of spatial domains to compute resources is referred to as "replication," as the mapping will replicate some domains across the resources. There are two main strategies for replication: static and dynamic. Static replication makes assignments when the program first begins and uses those assignments throughout execution. Dynamic replication changes assignments as the algorithm executes, in order to maintain load balancing. Both replication strategies aim to improve efficiency — they operate by replicating the spatial domains that have more particles, in order to distribute the workload more evenly across compute resources.

Dynamic replication is part of an overall approach for Monte Carlo transport. Each cycle of a Monte Carlo approach consists of three phases: initialization, tracking, and finalization. When incorporating a dynamic replication algorithm, the initialization phase executes the dynamic replication algorithm. The tracking phase does a combination of particle transport and communicating particles. Particle transport can operate in an embarrassingly parallel fashion, until MPI communication is required as particles move from one spatial domain to another (and thus need to be re-assigned to a compute resource that has that spatial domain). The finalization phase processes the distributed results of the tracking phase. Importantly, the initialization phase determines the performance of the tracking phase — if the domain assignments from the dynamic replication algorithm create balanced work for each compute resource, then all compute resources should complete the tracking phase at the same time, ensuring parallel efficiency.

Tracking is the computationally dominant portion of the algorithm. During tracking, each particle makes small advancements for short periods of time, and

each advancement is referred to as a "segment." The type of activity within a segment can vary, which affects the computational cost and duration of the advancement for a segment. In this paper the three relevant activities are: (1) collisions with the background material, (2) moving between mesh elements, and (3) moving to the end of the time step. Tracking concludes when each particle has advanced for a period equal to the overall cycle duration — if the overall cycle takes ΔT seconds, if a given particle advances via N segments for that cycle, and if each segment i advances for some time t_i seconds, then $\sum_{i=0}^{N-1} t_i = \Delta T$.

6.4 Related Works

Many works have studied spatial domain decomposition methods for Monte Carlo particle transport. The method was introduced by Alme, Rodrigue, and Zimmerman (2001), as they split a problem into a few parts, allowing for replications of spatial domains in order to parallelize the workloads while maintaining processor independence. Their proposed method was adopted by the Mercury simulation code and implemented in a production environment; Procassini et al. (2005) then provided empirical evidence for its efficacy. Spatial domain decomposition methods were further analyzed by Brunner and Brantley (2009); Brunner et al. (2006), who also contributed improvements for increasing scalability and improving performance overall. One of their important improvements for scalability was to add point-to-point communication, allowing processors in different spatial domains to communicate directly with one another. This was a change from a model where each spatial domain had a single processor which was in charge of all communication for that group of processors.

Work by M. J. O'Brien et al. (2013) introduced dynamic replication. Their scheme performed regular evaluation of parallel efficiency and then performed

load balancing when efficiency dropped below a specified threshold. M. O'Brien (2007) extended this work by adding a communication graph, which defined which processors can perform point-to-point communication during a cycle. Using these new communication graph algorithms, O'Brien et al. was able to successfully scale Mercury on LLNL's Sequoia supercomputer to over one million processors while maintaining good parallel performance. This work showed that keeping the load balance during particle communication within a cycle is important for scaling parallel performance. When particles were communicated to neighbors without considering load balance, a single processor could become bogged down with significantly more work — work which potentially could have been shared. Additional extensions to this work can be seen in other groups as well, such as with Ellis et al. (2019) who looked into additional mapping algorithms under specific conditions in the Monte Carlo transport code, Shift. Their work extends the communication graph concept by combining it with Monte Carlo variance reduction techniques to improve the overall efficiency for their use-cases.

While many works have focused on algorithmic improvements, many others have focused on evaluating load imbalance effects. In his PhD thesis, Romano (2013) expanded upon the concept of domain decomposition algorithms by providing new analytical understanding. In particular, Romano provided a basis for understanding the importance of load imbalance and being able to determine analytically the benefit of this method. Wagner et al. (2011) took a more empirical approach when studying load imbalance of reactor physics problems. They considered the problem of load imbalance stemming from spatial decomposition, and proposed new decomposition methods for handling this issue. Similarly, Horelik, Siegel, Forget, and Smith (2014) explored several spatial domain

decomposition methods and analyzed their effect on load imbalance. In summary, each of these groups identified load imbalance as a problem and proposed analysis and solutions that fit their specific needs.

As noted in the Motivation section, our closest comparator is a separate work from M. O'Brien et al. (2019). This work considered the problem of balancing particles in a given spatial domain among processors of varying speeds, but it did not consider domain decomposed meshes. As domain replication strategy is an important aspect to achieve performant algorithms, developing an algorithm that supports both heterogenous computing and domain decomposition is non-trivial and requires fresh investigation. This gap is the focus of our work.

6.5 Our Method

This section describes our novel dynamic replication algorithm for Monte Carlo transport. Our algorithm is optimized for heterogenous architectures — it assumes that individual computational resources will have different levels of compute power, and makes assignments based on that knowledge. Our algorithm consists of three steps:

1. **Assignment** (Section 6.5.1): identify how many times to replicate each spatial domain, and then assign those domains to compute resources.
2. **Distribution** (Section 6.5.2): partition the particles across compute resources.
3. **Mapping** (Section 6.5.3): build a communication graph between compute resources in order to communicate particles that have exited their current spatial domain during tracking.

6.5.1 Step 1: Assignment.

This step produces an assignment of compute resources to spatial domains, with the goal of making an assignment that

minimizes execution time. In particular, the number of particles per spatial domain varies, and so the goal is to replicate the domains with the most particles in order to assign a commensurate level of compute to each domain. The algorithm works by considering work and compute as proportions — if a domain has 10% of the particles, then that domain should be replicated so that it gets 10% of the compute resources. Further, if the assignments are effective, then all compute resources should complete at the same time during the tracking phase.

To make assignments, our algorithm needs to understand (1) how much work needs to be performed and (2) how capable the compute resources are. In both cases, we use results from the previous cycle, which we find to be a good representation for what work to expect in the next cycle. Explicitly, the total work for each domain is the number of segments to execute. We consider the per-domain work from the previous cycle as our estimate for the upcoming cycle. For compute rate, we consider how many segments per second each type of resource achieved. That is, we measure the average number of segments per second over all of the CPUs and the same for GPUs. Using past performance automatically accounts for variation in translating FLOPS to segments across hardware; where the FLOP ratio between a GPU and CPU may be 100:1, the ratio in average number of segments per second may be much lower, like 20:1.

Our algorithm depends on considering both work and compute in proportion to the whole, and we define three terms for ease of reference. Let PW_i be the proportion of work within spatial domain i. For example, if domain i has 10% of the total estimated work, then $PW_i = 0.1$. Further, let $PC\text{-}GPU$ and $PC\text{-}CPU$ be the proportion of total compute for a GPU and a CPU, respectively. For example, if a GPU can do 100 million segments per second, if a CPU can do 5

million segments per second, and if there are 4 GPUs and 20 CPUs, then the total capability is 500 million segments per second, and $PC\text{-}GPU = 0.2$ and $PC\text{-}CPU = 0.01$.

At the beginning of program execution, we assign each domain one GPU and one CPU. This ensures that every domain has "surge" capability in case the work assignment estimates are incorrect (which can happen when particles migrate from one domain to another at a high rate). Such surge capability prevents the worst case scenario — one compute resource takes a long time to complete its work, and the others sit idle. Further, one of these compute resources (either the CPU or GPU) can act as a "foreman" for its spatial domain. These foremen are bound to a spatial domain throughout program execution. When a compute resource is assigned a new spatial domain, it can get that domain from the appropriate foreman. The remaining compute resources can then be assigned to work on spatial domains dynamically.

Our assignment algorithm works in two phases. The first phase decides how many compute resources should be assigned to each spatial domain, and what type they should be. The second phase uses this information to make actual assignments to specific compute resources, being careful to minimize communication by keeping the same spatial domains on the same compute resources when possible.

The first phase employs a greedy algorithm, and is described in pseudocode below labeled "MakeGreedyAssignments." It begins by setting up an array variable that tracks how much work is remaining for each spatial domain ("RemainingWork") using the predicted work (PW_i) and taking into account the pre-allocated resources (one CPU and one GPU for each of the M spatial domains). The final step is to assign the remaining compute resources to spatial

domains. $NGPU$ is the number of GPUs, it begins by assigning the $NGPU - M$ available GPUs to spatial domains, one at a time. Each time, the algorithm first finds the spatial domain d with most remaining work, i.e., its evaluations takes into account that resources have been assigned previously. After the GPUs, it then makes assignments for each of the $NCPU - M$ available CPUs in a similar manner.

```
def MakeGreedyAssignments(M, NGPU, NCPU, PW, PCGPU, PCCPU):
   for i in range(M):
      WorkRemaining[i] = PW[i]
   # Account for preallocated resources
   for i in range(M):
      WorkRemaining[i] -= (PCGPU+PCCPU)
   NGPU -= M
   NCPU -= M
   # Replicate remaining compute resources greedily
   for i in range(NGPU):
      d = FindDomainWithMostWork(WorkRemaining)
      WorkRemaining[d] -= PCGPU
      AssignGPUToSpatialDomain(d)
   for i in range(NCPU):
      d = FindDomainWithMostWork(WorkRemaining)
      WorkRemaining[d] -= PCCPU
      AssignCPUToSpatialDomain(d)
```

All replication schemes nearly always have some load imbalance. Consider a problem with two spatial domains with equal amounts of particles ($PW_0 = PW_1 = 0.5$) and three GPU compute resources, C_0, C_1, C_2 where $PC\text{-}GPU = 0.333$. Then

C_0 and C_1 will be foremen, and the only question is whether to replicate domain 0 or 1 on C_2. Whatever the outcome, one domain will have a $WorkRemaining$ value of 0.167. In this example, it would be up to the foreman to carry out this extra work and it would be likely that the extra compute resources would be idle as it does so. Fortunately, these effects get smaller as concurrencies get larger. Also, the heterogeneous nature of compute helps on this front, as there are more resources (the CPUs) that are smaller (i.e., smaller values of $PC\text{-}CPU$) leading $WorkRemaining$ values being closer to 0 on the whole.

The second phase assigns specific compute resources. Every time a compute resource is assigned a new domain, it must retrieve this domain from its corresponding foreman, incurring a communication cost. So the goal of this phase is to repeat assignments between compute resources and domains. For example, if the output of the first phase indicates that domain d should have 3 GPUs, then the second phase checks to see if there are 3 GPUs that had d in the previous cycle. If so, then those GPUs should be assigned to d again for the current cycle, as this prevents unnecessary communication. Of course, as the number of compute resources applied to a domain increases, new compute resources must be located and communicate costs are inevitable.

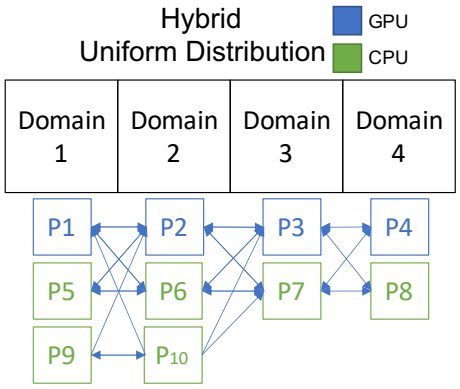
6.5.2 Step 2: Distribution. This step partitions the particles across compute resources. This partitioning must honor the spatial domain assignments, i.e., if particle P lies within spatial domain D, then the particle can only be assigned to compute resources that were assigned D. In our approach, we perform this partitioning relative to performance — GPU compute resources get more particles and CPU compute resources get less, and the proportion between them

corresponds to $\frac{PC\text{-}GPU}{PC\text{-}CPU}$. The remainder of implementation details follow trivially from previous work M. O'Brien et al. (2019).

6.5.3 Step 3: Mapping.

Mapping refers to establishing a communication graph between compute resources. This mapping is needed when particles exit their spatial domain. When this happens, they need to be sent from their current compute resource to another compute resource that is operating on their new spatial domain.

Figure 18. Result of our Map step with 4 spatial domains, 4 GPU compute resources, and 6 CPU compute resources. The square boxes show which domains neighbor (1-2, 2-3, 3-4).



In a domain replication environment, a poor communication graph can affect overall performance. For example, assume that domain d is replicated by K compute resources — C_0, C_1, ... $C_{(K-1)}$. One possible communication graph could instruct all other compute resources to send their particle entering d to C_0. This is bad: C_0 would spend more time doing communication than the other C_i resources and it also will end up with more particles to transport. Instead, a better mapping would lead to an even spread of particles between the C_i's.

For a given compute resource, our Map algorithm makes connections to all neighboring domains. It uses a round robin algorithm to prevent load imbalance, specifically:

$$index_A \ mod \ size_B = index_B \ mod \ size_A$$

where A is a list of resources from one domain and B is a list of resources from a second domain (see Figure 18). Our Map algorithm makes two connections for each neighboring domain d — one to a CPU compute resource that contains d and one to a GPU compute resource that contains d. Each connection also has a weighting which dictates the proportion of particles communicated. For our experiments, we set the weights to be proportional to their compute abilities (*PC-GPU* and *PC-CPU*), i.e., a GPU resource would be sent many more particles than a CPU resource. That said, exploring different weights would be interesting future work, in particular weights where CPUs get more particles.

6.6 Experiment Overview

This section provides an overview of our experiments, and is organized into three subsections. Subsection 6.6.1 describes the hardware and software used for our experiments. Subsection 6.6.2 describes the factors we vary to form our set of experiments. Finally, Subsection 6.6.3 describes the measurements we use to evaluate our results.

6.6.1 Hardware and Software. Our experiments were run on LLNL's RZAnsel supercomputer. This platform has two Power 9 CPUs (22 cores per CPU, of which 20 are usable), 4 Nvidia Volta GPUs (84 SMs per GPU), and NVLink-2 Connections between the sockets on each node. In addition, there are a total of 256 GB of CPU memory and 64 GB of GPU memory per node *Livermore Computing Center High Performance Computing: RZAnsel* (2020). For software, we used Imp, by P. Brantley et al. (August, 2019), a Monte Carlo code that solves time-dependent thermal x-ray photon transport problems.

6.6.2 Experimental Factors. Our experiments vary two factors: workload (11 options) and hardware configuration (3 options). We ran the cross product of experiments, meaning 33 experiments overall.

Workloads: our 11 unique workloads consisted of three distinct problems ("Crooked Pipe," "Holhraum," and "Gold Block"), with one of those problems ("Gold Block") having nine different variations. Details for each of the three distinct problems are as follows:

- **Crooked Pipe:** a problem that simulates transport through an optically thin pipe with a U-shaped kink surrounded by an optically thick material. The Crooked Pipe problem is load imbalanced since particles are sourced into the leading edge of the pipe, causing spatial domains that contain this region to have a much higher amount of work per cycle than the others. This is a common test problem in the Monte Carlo photon transport community as well as an excellent driver for testing load balancing methods.
- **Hohlraum:** a problem that simulates the effects of Lawrence Livermore's NIF laser on a gold hohlraum. Particles in this problem start in an incredibly hot gold wall and then propagate throughout the mostly hollow interior, colliding with a central obstruction as well as the surrounding gasses. This problem starts out very load imbalanced with most work in the hot region.
- **Gold Block:** a homogenous test problem that simulates a heated chunk of gold. This problem is a solid cylinder of gold with reflecting boundary conditions. Since this problem is a homogeneous material with reflecting boundary conditions, we can modify the length scale of the problem in order to change the ratio of the number of collision segments with the number of total segments by changing the number of mesh element crossing segments

and leaving all else fixed. We use this length scaling to create a total of 9 configurations, with an unscaled version at the center we refer to as the Base Gold Block. Specifically, we took our Base Gold Block problem and halved the length scale 4 times consecutively, and similarly doubled the length scale 4 times consecutively to create these configurations. The goal with this scaling is to understand performance with respect to the percent of time performing collisions segments versus other segment types.

Hardware configurations: we ran each of the workloads with:

- **Hybrid:** our algorithm, scheduled with both GPUs and CPUs.
- **CPU-Only:** scheduled using only CPU resources
- **GPU-Only:** scheduled using only GPU resources

For the **CPU-Only** and **GPU-Only** tests, we were able to perform experiments using our algorithm, since our algorithm simplifies to be the same as predecessor work when the resources are homogeneous. Further, all experiments were run on 4 nodes, meaning we used: for CPU-Only 160 CPU resources, for GPU-Only 16 GPU resources, and for Hybrid 144 CPU resources + 16 GPU resources.

6.6.3 Measurements. To analyze our results, we considered three types of measurements:

- **Throughput** defines the number of segments, on average, that a processor will be able to process in one second. This metric is used to compare application performance in a consistent manner, regardless of hardware or software configuration.

- **Segment Counters** divide the segments into the three different activity types considered in this paper (see Section 6.3). Specifically, these counters

count the total number of times each type of segment has occurred across all segments in the simulation. Segment counters are useful for understanding how performance varies with respect to different segment types.

– **Efficiency** determines the success of a load balance algorithm. For each domain i, we calculate the ratio of the compute resource applied to that domain (sums of *PC-GPU* and *PC-CPU* for the assigned C_i's) and work for that domain (PW_i). For example, a given domain may have 8% of the compute resources and 10% of the total work, for a ratio of 0.8 or an efficiency of 80%. Our efficiency metric is the minimum of these ratios over all domains, meaning 1.0, 100%, is a perfect score (compute resources applied perfectly in proportion to work for all domains) and less than 1.0 indicates the inefficiency — a score of 0.5 indicates that one domain has been given half the resources it needs, i.e. it has an efficiency of 50%.

6.7 Results

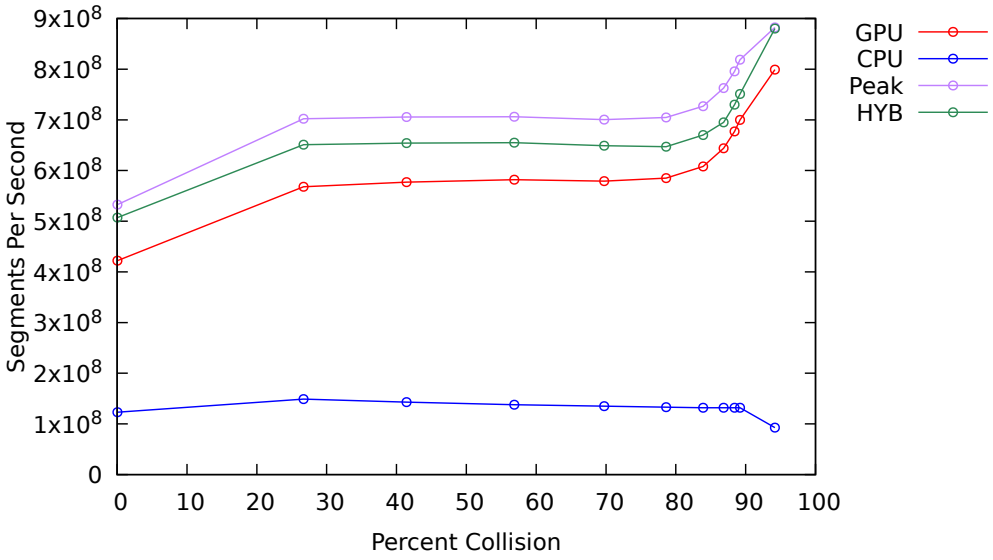
Our results are organized into two parts:

– Section 6.7.1 evaluates the performance of our overall heterogenous algorithm.

– Section 6.7.2 evaluates the efficacy of our load balancing algorithm.

6.7.1 Algorithm Performance. Figure 19 shows the performance results for our 33 experiments. This figure contains a line for "peak" performance. This does not represent actual experiments, but rather a theoretical analysis of the potential peak speedup from using both CPUs and GPUs. This line was calculated by taking the GPU performance and adding 90% of the CPU performance (since some CPUs are needed to manage GPUs in a heterogeneous environment, specifically 36 of the 40 CPU cores were used for computation, while the remaining

Figure 19. Plot showing throughput (segments per second) as a function of what proportion of the segments were of the type "collision." It plots four lines, one for each of our three hardware configurations, and one for a theoretical "peak" configuration (described in Section 6.7.1). Each of the dots come from our workloads — the left-most (∼0% collisions) come from the Crooked Pipe problem, the right-most (∼100% collisions) come from the Hohlraum problem, and the remainder come from variations of the Gold Block problem.



4 managed GPUs). This peak line should be viewed as a "guaranteed-not-to-exceed" comparator.

One important finding is on the potential of heterogenous computing for this problem. While the CPUs have only 3% of the FLOPs of the GPUs, their performance (i.e., throughput) is much better than 3%. CPUs have 26.2% of the throughput for the Crooked Pipe problem, 20.4% for the Base Gold Block problem, and 10.4% for the Hohlraum problem. In all, this provides important evidence that including CPUs can be much more beneficial than a basic FLOP analysis. Of course, this potential can only be leveraged with an effective algorithm.

With respect to actual achieved performance, our heterogenous algorithm ("Hybrid") performed quite well. It was 20.4% faster than GPU-only for the Crooked Pipe problem, and approximately 10% faster for the other problems.

Relative to the peak line, our algorithm achieved 95.1% for Crooked Pipe, 91.8% for Base Gold Block, and 99.6% for Hohlraum. The performance is greatest where the amount of collision segments is dominant, which is also where there is a larger amount of compute used by the resources. In this region, the CPUs are less valuable, but still more valuable than the hardware specification predicts. On the other end of the spectrum, where there are less collisions and more mesh element crossing segments, the compute is lower, and the GPUs are less performant. This enables the CPUs to provide an even greater benefit overall.

Table 11. This table shows the efficiency for our three workloads over a full program execution. Minimum efficiency represents the worst assignment over all compute resources and cycles: for one cycle of the Crooked Pipe problem, there was a compute resources which had about 20% too much work to finish on time with the other compute resources. Maximum efficiency speaks to the best cycle: for one cycle of the Crooked Pipe problem, the most underpowered compute resource had only 0.1% too much work. Average efficiency speaks to the behavior across cycles: (1) for each cycle, identify the most underpowered compute resource and calculate how much extra work it has, and (2) take the average over all cycles of the extra work amounts. For the Crooked Pipe problem, the average efficiency is 99.55%, meaning that the average slowdown for completing a cycle due to load balancing was <0.5%.

Problem	Minimum Efficiency	Maximum Efficiency	Average Efficiency
Crooked Pipe	81.76%	99.92%	99.55%
Base Gold Block	81.76%	99.9%	99.90%
Hohlraum	81.76%	86.16%	85.80%

6.7.2 Load Balance Efficiency. Table 11 plots efficiency results for our three workloads. On the whole, the minimum efficiency values for these workloads are low. That said, these conditions occur in the first few cycles, as these cycles do not have a history of performance to base their load balancing decisions on. For Crooked Pipe and Base Gold Block, the average efficiencies indicate that good load balance is achieved quickly and maintained throughout the run. The

Hohlraum problem had worse efficiency. This was because one domain was a "hot spot" — it had much more work than the other domains. This topic is explored further in the following subsection.

6.7.3 Surge Capability. The Hohlraum workload demonstrates the value in our "surge capability" (ensuring that one GPU and CPU are assigned to each domain). This workload had 4 domains, and domain 1 had the majority of particles, to the point meriting assignment of every non-foreman compute resource. That said, during a compute cycle, domain 1's particles stream out rapidly into neighboring domains. Our surge capability ensured extra compute resources were allocated, and this made a 3X performance improvement for this case. Figure 20 has more details on this comparison, with Gantt charts that show behavior within a cycle. Finally, the "surge" allocation had no impact on the other two problems since their work was more balanced, and they would have received those resources anyway.

6.8 Conclusion

In this chapter, we introduce a novel load balancing algorithm which can efficiently partition heterogeneous compute resources across domains. We demonstrate results using this algorithm, in a production Monte Carlo photon transport code, running a variety of workloads. This work was motivated by the performance difference seen in practice between Monte Carlo transport codes running on CPUs and GPUs when compared with the ratio of the available FLOPs. Our algorithm demonstrated up to a 20% performance benefit, which is much greater than the 3.7% predicted by solely looking at the ratio of FLOPs. Additionally, our algorithm achieves 85% to 99% load balancing efficiency on the problems demonstrated.

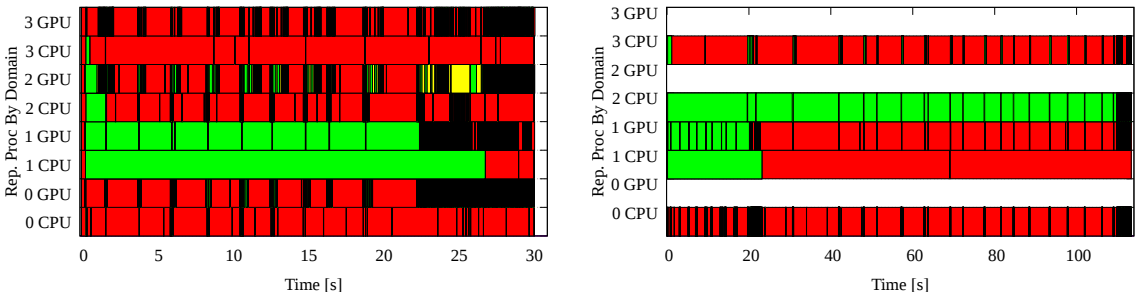


Figure 20. Gantt charts for a single cycle of the Hohlraum workload. The left Gantt chart corresponds to our algorithm, and completes in 30s. The right Gantt chart is a variant of our algorithm where there is no minimum compute allocation (i.e., the "surge capability" is disabled). This variant took 113s, 3.8X slower. The Gantt plots show an evolution over time per compute resource, with red representing "idle" time, yellow representing communication time ("MPI Send/Recv"), and green representing time spent tracking particles. The blank spaces in the right chart occur because there is no compute resource assigned to that domain, for example no GPU resource for domain 3. Finally, these Gantt charts show only the first CPU and first GPU for a domain, and the other compute resources are not plotted. In particular, the remaining compute resources in the left Gantt chart (our algorithm) are doing tracking (green) at a high rate, consistent with the overall efficiency of 85% — some of the worst performers for this workload (domains 0, 2, and 3) are being plotted.

CHAPTER VII

PERFORMANCE AT SCALE

The work presented in this chapter comes from a publication that is currently in preperation. I was the primary contributor in developing the tests, running experiments, and writing. Dr. Patrick Brantley, Dr. Matthew O'Brien, and Dr. Hank Childs provided ideas and feedback throughout the development process and also assisted in editing the content.

7.1 Motivation

In this dissertation many ideas are presented that provide a guide to developing an effective Monte Carlo transport algorithm for many-core architectures. That said, it is important to confirm that the techniques will work in real-world settings. In particular, techniques may work well at lower concurrency, but be less effective at scale, or techniques may not work in combination. The goal of this chapter is to review the combined effects from using these ideas on a fully featured application when scaled on a GPU based supercomputer. In order to evaluate the effectiveness of these ideas, experiments are used which cover a wide variety of workloads. This evaluation provides evidence for which decisions, many of which were made in mini-apps, are applicable when considering fully featured applications and MPI scaling.

This chapter explores the real-world viability issue via fully featured Monte Carlo transport applications with additions from each research topic. First, the research in mini-apps from Chapter III suggests that history-based Monte Carlo should work on GPUs, but that large complex kernels could benefit from an event-based approach. As a result, the experiments will focus primarily on the history-based approach to map out its viability in a full scale application, though initial

results with an event-based approach are explored as well. Second, the Thin-Threads threading model described in Chapter IV is used for both CPU and GPU results. Third, the concept of variable replication of tally data, explained in Chapter V, is used for the balance tallies — frequently used single value tallies — with 16 replications used to reduce contentious atomics. Fourth, the hybrid CPU+GPU approach, described in Chapter VI, is compared with CPU only and GPU only approaches in order to better understand the workloads and scales where the hybrid approach is beneficial and/or viable.

The remainder of the chapter is organized as follows. Section 7.2 (Experiment Overview) gives an overview of the hardware, software, and experiments that are used to generate the data. Section 7.3 (Results) presents the evaluation of each experiment. Finally, section 7.4 (Conclusion) summarizes the important factors presented in the Results section, showing the viability of the approaches presented in this dissertation.

7.2 Experiment Overview

This section provides an overview of the experiments for this study. Subsection 7.2.1 provides an overview of the hardware and software used to generate the results. Subsection 7.2.2 describes the set of measurements taken and presented during the study. Subsection 7.2.3 describes the study configuration, including the factors varied to form these configurations. Subsection 7.2.4 provides a description for each problem used in this study.

7.2.1 Hardware and Software.

All of the results gathered in this chapter were generated on LLNL's RzAnsel supercomputer, described by *Livermore Computing Center High Performance Computing: RZAnsel* (2020). This platform has 54 compute nodes,

with each compute node composed of 2 IBM Power9 CPUs (each of which has 22 cores, 20 usable) and 4 Nvidia Volta (V100) GPGPUs. This hardware gives a combined peak performance of 1,570 TFLOPs, with the CPUs providing 58 TFLOPs and the GPUs providing 1,512 TFLOPs.

The software used to generate the results in this chapter are the LLNL production codes Mercury and Imp, described by *Mercury* (2019) and P. Brantley et al. (August, 2019), respectively. Mercury and Imp are Monte Carlo transport codes that share infrastructural source code for all general functionality. That said, each code has its own code-specific implementations for calculating the specifics of physics equations. Specifically, Mercury handles the neutron, light element charged particle, and gamma photon transport capability, while Imp handles the thermal x-ray photon transport capability. Additionally, Mercury can run using a continuous energy model or a multi-group energy model — where energies are stored in groups but treated as continuous values when tracking (not utilizing a specialized multi-group treatment).

7.2.2 Measurements.

The primary measurement used to understand the impact of each problem is simulation wall-clock time, specifically, the time spent in the cycle tracking portion of the Monte Carlo transport algorithm. This data will be presented in three distinct ways. First as time associated with each collected data point. Second, as speedup which is calculated by treating the CPU only experiments as a baseline and dividing by this time. Third, as scaling efficiency which is calculated for weak and strong scaling studies. We use this formula for weak scaling efficiency:

$$Weak_{eff} = \frac{T_1}{T_n} \times 100$$

and this formula for strong scaling efficiency:

$$Strong_{eff} = \frac{T_1}{N \times T_n} \times 100$$

where T is time and N is number of nodes.

In order to understand a large problem space, we will vary multiple factors in this experiment. These factors are described below in subsection 7.2.3 (Factors).

7.2.3 Factors.

The workloads for this study were generated by varying four primary factors: problem, node count, workload, and approach. Each of these factors enables understanding the impact of underlying phenomena. In all we ran a total of 162 experiments, with 144 of the experiments running with Imp and 18 running with Mercury. The 144 Imp experiments covered the following cross product of these factors:

- 2 problems: Crooked Pipe and Hohlraum.
- 6 node counts: 1, 2, 4, 8, 16 and 32 nodes.
- 4 workloads: 2.5, 5, 10, and 20 million particles per node.
- 3 approaches: CPU only, GPU only, and Hybrid CPU+GPU.

The 18 Mercury experiments covered the cross product of these factors:

- 1 problem: Godiva in water.
- 6 node counts: 1, 2, 4, 8, 16 and 32 nodes.
- 1 workload: 20 million particles per node.
- 3 approaches: CPU only, GPU only history-based, and GPU only event-based.

The following subsections will describe the importance of each of these factors as well as how varying this factor is accomplished.

7.2.3.1 Problem.

The first important factor to vary in our experiments is the problem that we are solving. Varying this factor allows for comparisons in performance between neutron transport and photon transport physics routines, as well as comparing streaming particle performance with collisions physics focused problems. Mercury is used to solve neutron transport problems and Imp is used to solve photon transport problems. Since both of these codes rely on shared infrastructural source code, the mesh based tracking portion of the code will be the same for all problems. However, Imp and Mercury each implement separate collision physics routines to solve their specific problems, providing a variety in code paths and levels of complexity to explore.

Details for each of the test problems can be found in subsection 7.2.4 (Problem Descriptions).

7.2.3.2 Node Count.

The second factor to vary in our experiments is the node count. Varying this factor modifies the impact that MPI has on performance and can highlight areas where algorithms rely on MPI communication. The ability for algorithms to scale efficienctly is critical for use in a supercomputer environment and this factor is crucial to understanding this impact.

7.2.3.3 Workload.

The third important factor is workload, which can greatly affect performance. When used in combination with varying the node count workload

studies are used to gain an understanding of the weak scaling and strong scaling capabilities of an application.

A weak-scaling study is generated by keeping the number of particles per node fixed as the number of nodes in use is increased. A strong-scaling study is generated by keeping the total number of particles fixed as the number of nodes is increased. In order to produce the strong scaling results in this study, we ran four weak scaling studies at different fixed sizes: 2.5, 5, 10, 20 million particles per node. In doing this we generate a series of smaller strong scaling results which span the scale of the problem.

7.2.3.4 Approach.

The final factor to vary is which approach to use, both hardware and software. Varying these approaches is critical to understanding the impact that all of our changes have made to many-core Monte Carlo transport applications. Hardware approaches include: CPU only, GPU only, or a Hybrid CPU+GPU approach. A CPU only approach provides a baseline comparator for evaluating performance gains. A GPU only approach is the result of most of the work presented in this dissertation and provides evidence of the success of many of the algorithms. A Hybrid CPU+GPU approach (referred to as the Hybrid approach) is an extension of the GPU only method that provides a possibility for increased performance and which merits further study. For each of these hardware approaches there are two possible software approaches. Software approaches include: history-based or event-based tracking algorithm. When discussing problems in Imp we will only look at the history-based software approach. When discussing problems in Mercury we consider both history and event-based software approaches for the GPU only hardware approach.

7.2.4 Problem Descriptions.

This section describes the specific problems used to generate results. Each of these subsections also corresponds to a section in the results section describing the results running this problem.

7.2.4.1 Crooked Pipe.

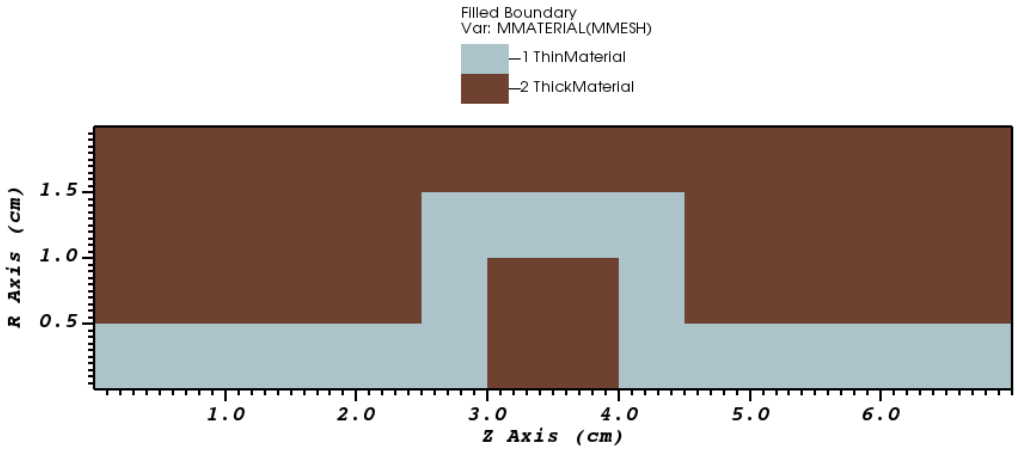
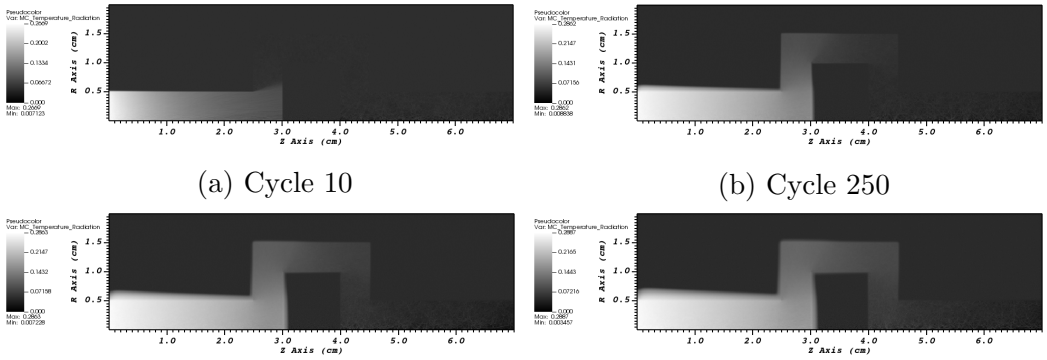


Figure 21. The Crooked Pipe 2D test problem where the brown region represents the optically thick material and the blue-grey region represents the optically thin region.

The Crooked Pipe test problem, initially described by Graziani and LeBlanc (2000), simulates transport through an optically thin pipe with a U-shaped kink surrounded by an optically thick material. Photons are sourced into the leading end of the pipe and travel the interior of the pipe easily. These same photons collide frequently with the walls of the pipe and when inside the surrounding thick material. Figure 21 gives a visual representation of this problem in 2D.

The Crooked Pipe problem is inherently load imbalanced due to the particles sourced into the leading edge of simulation. Figure 22 shows the radiation temperature of this problem at four separate cycles, highlighting this load imbalance. This causes the spatial domains that contain this source region



(a) Cycle 10 (b) Cycle 250

(c) Cycle 750 (d) Cycle 1000

Figure 22. The radiation temperature for the Crooked Pipe test problem at four points in time: (a) Cycle 10 or 9.00e-9[s], (b) Cycle 250 or 2.49e-07[s], (c) Cycle 750 or 7.49e-07[s], (d) Cycle 1000 or 9.99e-07[s]. In this problem color represents how hot the material is with white indicating high temperature, light-grey indicating a moderate temperature, dark-grey indicating low temperature and black indicating baseline temperatures.

to have a much higher amount of work per cycle than the others, thus requiring that domain decomposed algorithms manage load imbalance well in order to not become bogged down on this problem. This is a common test problem in the Monte Carlo photon transport community as well as an excellent driver for testing load balancing methods.

7.2.4.2 Hohlraum.

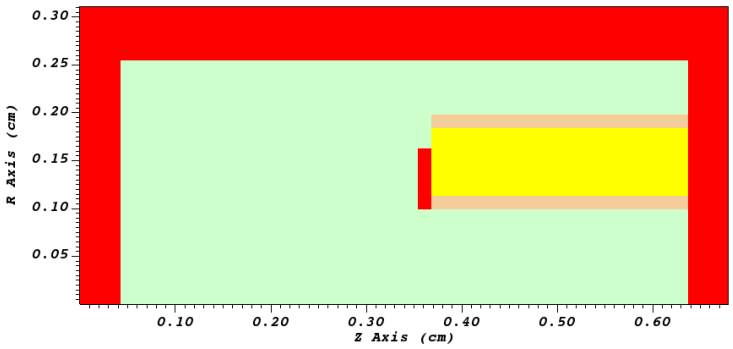
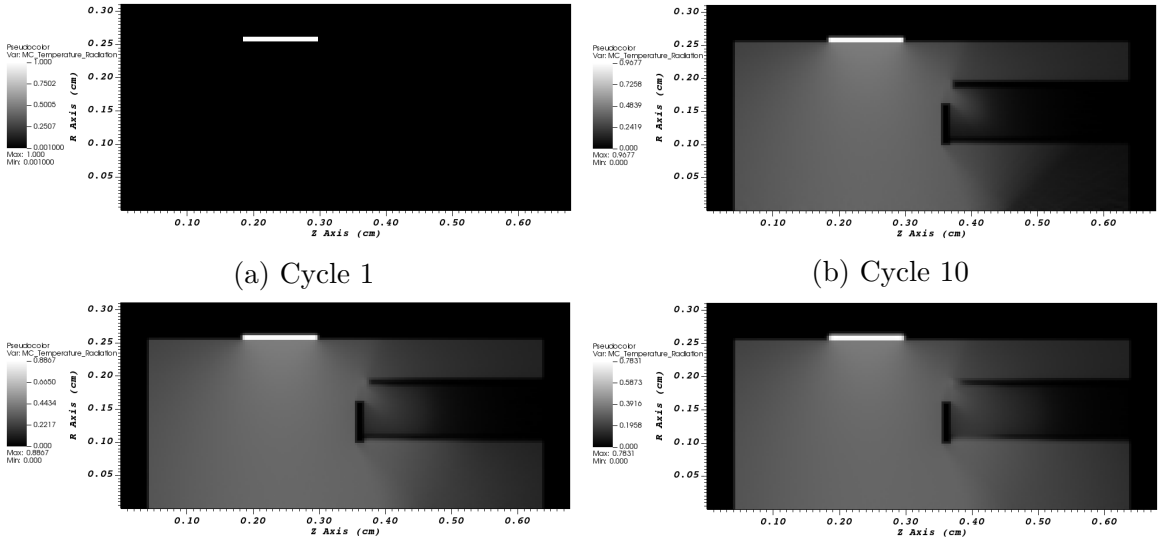


Figure 23. The Hohlraum test problem. The red region represents gold, tan represents tantalum, yellow represents a silicon foam, and light green represents helium gas.



(a) Cycle 1

(b) Cycle 10

(c) Cycle 20

(d) Cycle 30

Figure 24. The electron temperature for the hohlraum test problem at four points in time: (a) Cycle 1 or 0.000000e+00[s], (b) Cycle 10 or 8.435049e-11[s], (c) Cycle 20 or 3.147996e-10[s], (d) Cycle 30 or 9.046136e-10[s]. In this problem color represents how hot the material is with white indicating high temperature, grey indicating moderate temperature, and black indicating low temperature.

The hohlraum test problem, described by Yee, Olivier, Southworth, Holec, and Haut (accepted (2021)), simulates the effects of Lawrence Livermore's NIF laser on a gold hohlraum. The NIF laser targets the inner wall of the hohlraum causing it to rapidly heat up and emit x-ray photons. In order to simulate this effect, particles in this problem start by being thermally sourced from a hot spot in the gold wall of the hohlraum. These particles then propagate throughout the interior of the hohlraum and collide with a central obstruction or the surrounding gasses that fill the hohlraum. Figure 23 shows the structure of this test problem. This problem starts out very load imbalanced with most work in the hot region and over time starts to balance out as particles move through the problem heating up other regions while the hot spot cools. Figure 24 shows the electron temperature at four differing cycles in order to demonstrate the workload distribution.

7.2.4.3 Godiva In Water.

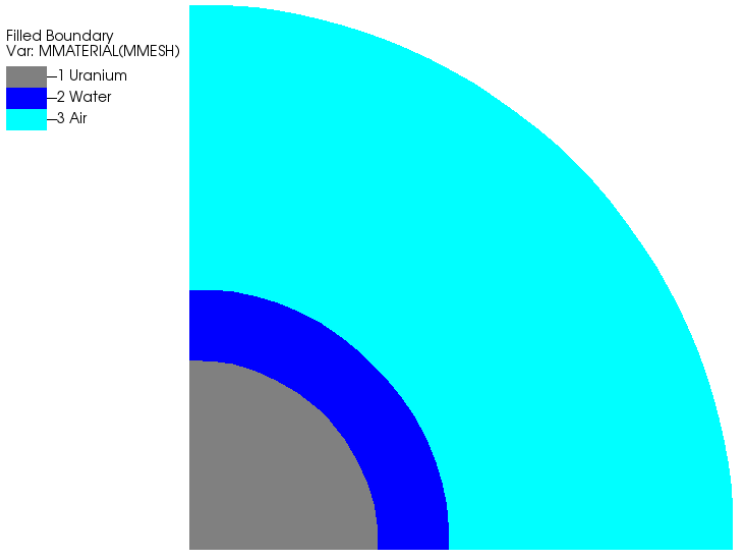


Figure 25. The Godiva in water test problem configuration. The grey region is uranium, the blue region is water, and the light-blue region is air. Outside the blue region is a vacuum boundary condition. This is a 2D slice of a 3D problem.

The Godiva sphere is a well understood and commonly used benchmark problem for neutron transport applications. The Godiva in water test problem is a small variation on this problem that surrounds the sphere of uranium with water and air in order to determine its new criticality, and is defined by Cullen et al. (2003). In order to calculate criticality we will use the static-k method. Figure 25 shows this setup on a 2D cross section for an octant of the full test problem. An octant of the full 3D test problem is simulated using reflecting boundary conditions, creating results for the spherical geometry. In order to understand the aggregate movement of neutrons, scalar flux values are computed on the problem mesh. Figure 26 shows the underlying scalar flux quantities for two of the 230 energy groups used in our calculations.

(a) Energy Group 150 (b) Energy Group 100

Figure 26. This data shows the scalar flux values for the Godiva in water test problem after one cycle. This calculation was run with a total of 230 energy groups and the values of scalar flux for two energy groups are displayed.

7.3 Results

This section describes results from the Imp and Mercury evaluations. Subsection 7.3.1 (Imp Crooked Pipe Analysis) examines Imp's Crooked Pipe problem through analysis of weak and strong scaling, scaling efficiencies, and speedup evaluations. Subsection 7.3.2 (Imp Hohlraum Analysis) examines Imp's Hohlraum problem through analysis of weak and strong scaling, scaling efficiencies, and speedup evaluations. Subsection 7.3.3 (Mercury Godiva In Water Analysis) examines Mercury's Godiva in water problem through analysis of weak and strong scaling, scaling efficiencies, and speedup evaluations. Finally, subsection 7.3.4 makes further comparisons between Mercury and Imp performance on these various problems and configurations.

7.3.1 Imp Crooked Pipe Analysis.

The Crooked Pipe problem in Imp is described in detail in subsubsection 7.2.4.1 (Crooked Pipe). There are two primary characteristics of interest that define this problem. Firstly, there are significantly more mesh element

crossing events than collisions events. Secondly, the problem is load imbalanced with more work in the domain that contains the source.

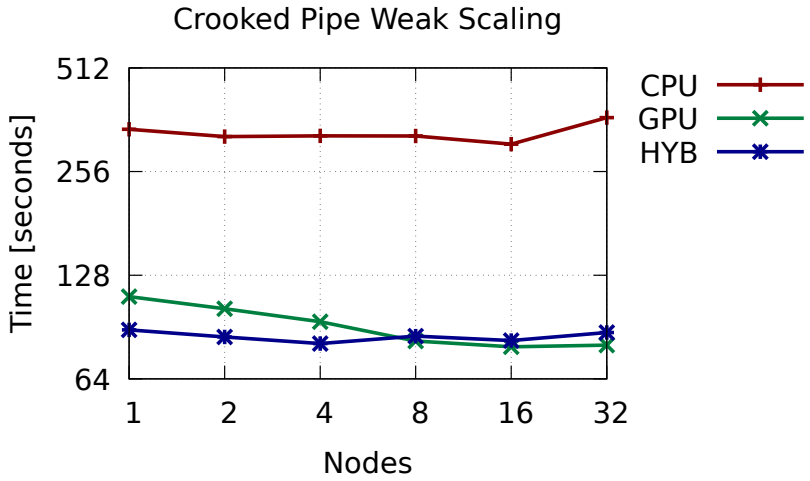


Figure 27. A weak scaling study of the Crooked Pipe problem on 1 to 32 nodes using 20 million particles per node. Results are shown that compare the CPU, GPU, and Hybrid approaches.

The first step to understanding the performance characteristics of this problem is to look at a weak scaling study comparing CPUs, GPUs, and the Hybrid approaches. Figure 27 shows the weak scaling results when running the Crooked Pipe problem with four domains and 20 million particles per node on up to 32 nodes. In this plot, a horizontal line represents perfect weak scaling. This plot shows that the CPU results have excellent weak scalability — their performance exceeds perfect scaling until the 32 node mark. At this point the plot starts to trend upwards, showing the introduction of minor inefficiencies at scale. Interestingly, the GPU results continue to improve up to the 32 node mark. This can be explained by looking at the load balancing characteristics of this problem. Firstly, with 4 GPUs per node, and 4 domains, there is no load balancing occurring with a single node. Further, it makes sense that the load balancing efficiency would

improve as more resources can be devoted to load balancing the problem. At 32 nodes, there is no longer continued improvement over 16 nodes, indicating that scaling inefficiencies are increasingly likely to emerge at higher and higher scales. Finally, the Hybrid results show a combination of the two previous patterns. The Hybrid approach does not gain as much of a benefit from increased number of GPUs as the GPU only approach does, but it starts out with better results at low scales. At higher scales, the CPUs are affecting the performance of the Hybrid approach in a negative way, with 8 nodes and above performing worse than the GPU only approach.

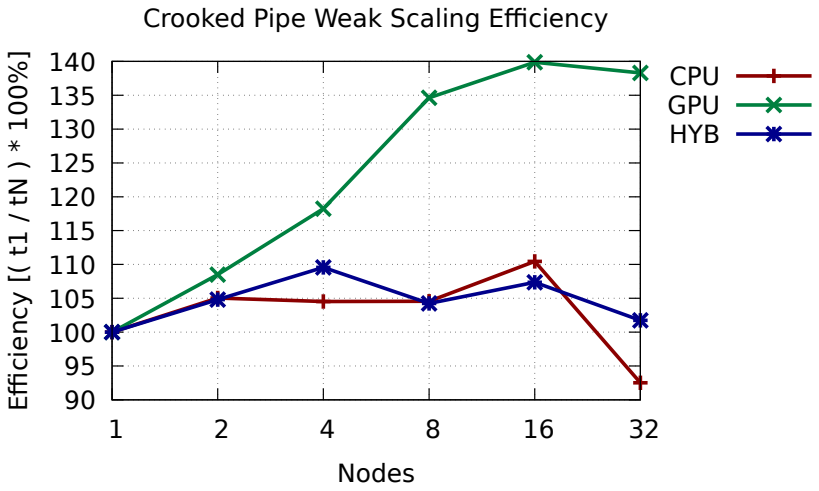


Figure 28. Efficiency for weak scaling study of the Crooked Pipe problem on 1 to 32 nodes using 20 million particles per node. Results are shown that compare the CPU, GPU, and Hybrid approaches.

Further understanding these weak scaling results requires evaluating the efficiency; these efficiency results are plotted in Figure 28. Interestingly, although efficiency often drops as the number of nodes increases, this does not happen for these experiments. For example, as mentioned previously, the GPUs show a significant performance improvement at node counts up to 16 nodes. This plot

makes it very clear that from 16 to 32 nodes is the turning point where increasing nodes no longer is beneficial to performance because of decreased efficiency due to load balance. Additionally, the CPU and Hybrid approaches behave much more similarly, which is most likely due to them containing the same number of MPI ranks — scaling effects in the code are affecting them in a similar way. The Hybrid approach has an additional benefit of maintaining its efficiency at around 100% when both the CPU and GPU results show more dramatic variations.

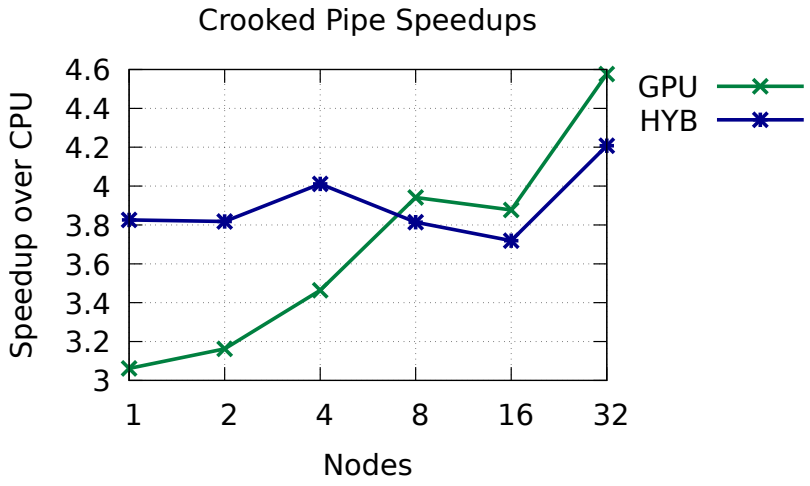


Figure 29. Crooked Pipe speedup of GPU and Hybrid approaches over CPU only approach

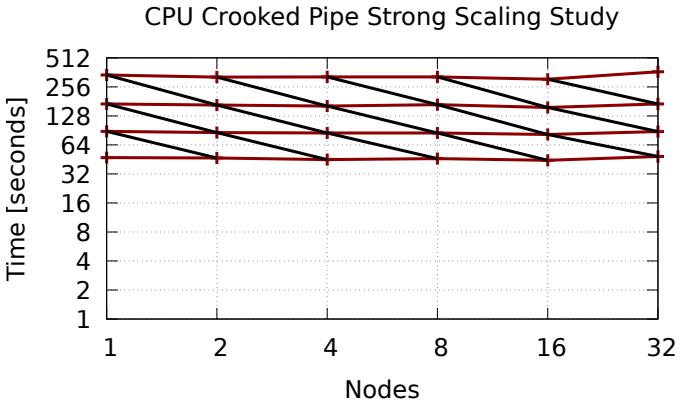
Speedup is a final way to analyze the results from this weak scaling study. Figure 34 gives the speedup of the GPU only and Hybrid approaches over the CPU only approach. From this plot we can see that the Hybrid approach is clearly faster when the GPUs are not able to load balance well — i.e., when the number of GPUs is close to the number of domains. This plot also shows that while the Hybrid approach does not maintain its lead, it still closely follows the GPU curve at higher node counts. This tells us that a slight improvement to the Hybrid approach could

recover this performance difference, allowing it to always be more efficient than GPU only for this problem.

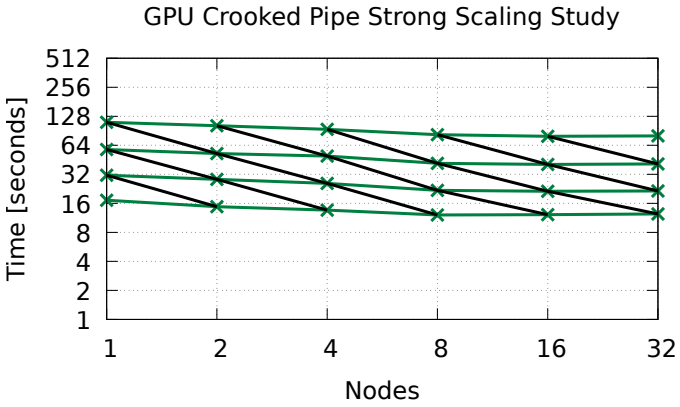
Another form of analysis is to look at the strong scaling capability of Imp. In order to generate studies for strong scaling results, four weak scaling studies were used. By using four weak scaling studies to produce these results a partial weak/strong scaling study can be observed. In this study, horizontal direction indicates a weak scaling study and diagonal direction indicates a strong scaling study.

Figure 30 provides the partial strong scaling study results for CPU, GPU, and Hybrid approaches, while Figure 31 plots the data for the efficiencies of this study. These results show that, firstly, the weak scaling of each approach is close to a perfect horizontal line, indicating good weak scaling efficiency. The only outlier to this case is that the CPU 16 to 32 node jump shows a higher time, and the additionally, the Hybrid lower particles per node show increasing lines up to 32 nodes. The strong scaling results look quite good, and the efficiencies agree for most cases. All of the strong scaling lines maintain a downward linear slope, indicating good efficiency, and the efficiency plots corroborate this with most maintaining near the 100% mark. The only outlier to this data is that the Hybrid approach shows poor strong scaling characteristics. This quick drop in efficiency can be explained by the increase in computing power of the Hybrid approach when compared to GPU or CPU approaches. Since the GPUs require a large amount of work to remain saturated, decreasing the amount of work per node, but also including CPUs to take some of the work, makes the Hybrid case show this behavior.

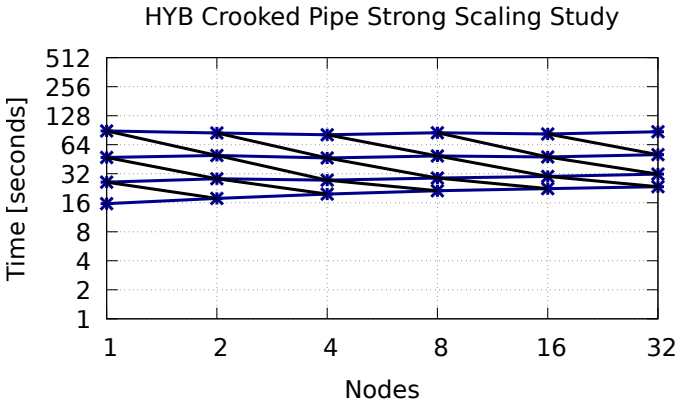
7.3.2 Imp Hohlraum Analysis.



(a) CPU approach

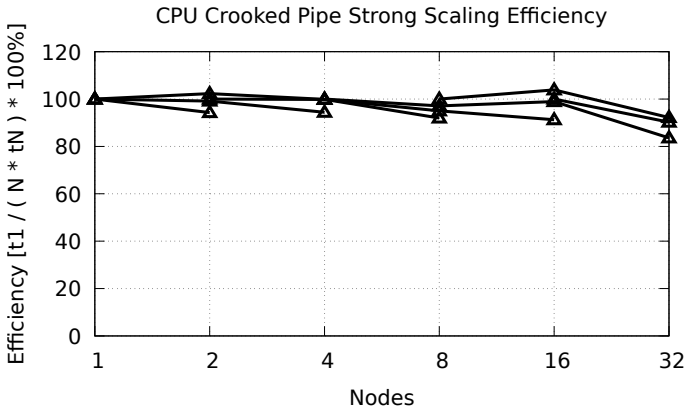


(b) GPU approach

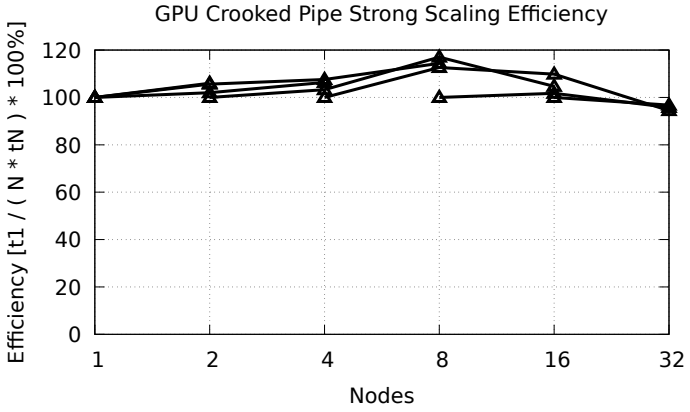


(c) Hybrid approach

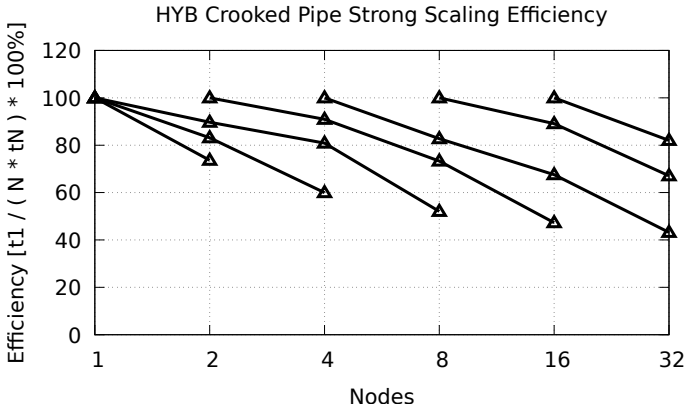
Figure 30. A weak/strong scaling study of the Crooked Pipe problem using (a) CPU, (b) GPU, and (c) Hybrid approaches.



(a) CPU approach



(b) GPU approach



(c) Hybrid approach

Figure 31. Efficiency of each strong scaling line from each of the studies done for (a) CPU, (b) GPU, and (c) Hybrid cases.

The Hohlraum problem in Imp is described in detail in subsubsection 7.2.4.2 (Hohlraum). There are three primary characteristics of interest that define this problem. Firstly, there are significantly more collision events than mesh element crossings. Secondly, the problem is load imbalanced with more work in the domain that contains the hot spot, but the work load also diffuses over time. Thirdly, the problem has a variable time step over the first 30 cycles and a fixed time step after 30 cycles. This leads to different amounts of work occurring each cycle and is a common method for managing complex interactions in Monte Carlo transport problems.

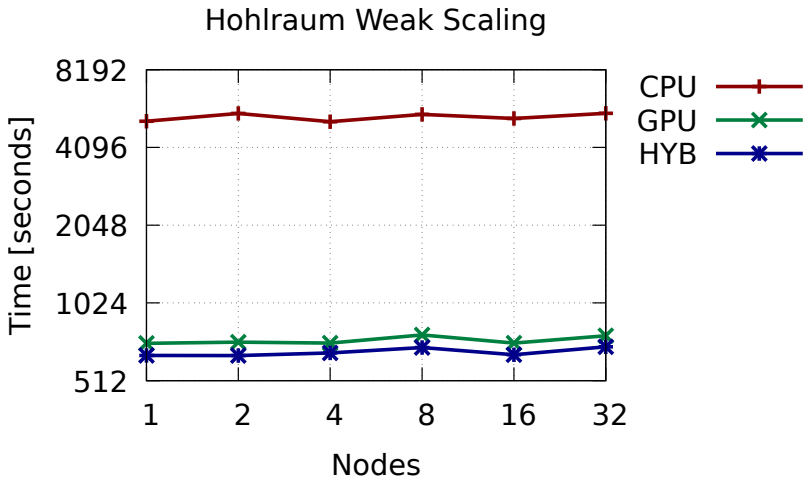


Figure 32. A weak scaling study of the Hohlraum problem on 1 to 32 nodes using 20 million particles per node. Results are shown that compare the CPU, GPU, and Hybrid approaches.

Figure 32 shows the weak scaling results for running this problem with 20 million particles per node and up to 32 nodes. These results show weak scaling plots that are nearly perfectly flat. This means that the weak scaling is happening nearly perfectly. While there are some slight variations in the 8 and 32 node cases

where the time increases slightly, there are no indications of a bad scaling curve at this scale.

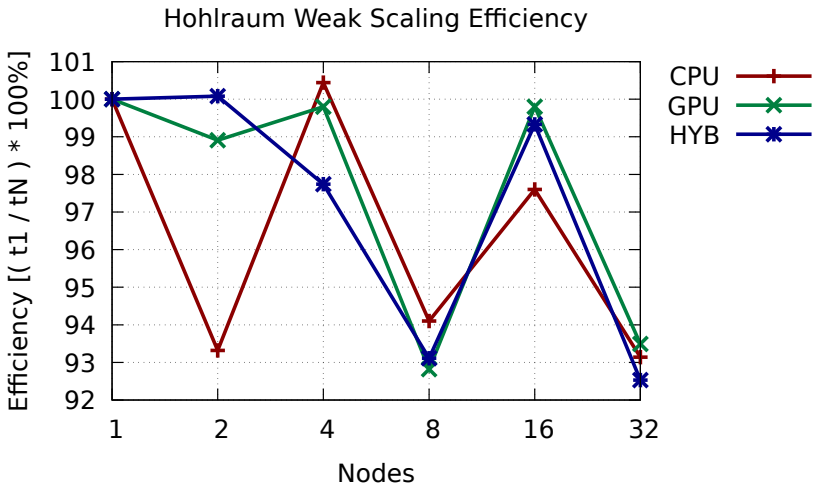


Figure 33. Efficiency for weak scaling study of the Hohlraum problem on 1 to 32 nodes using 20 million particles per node. Results are shown that compare the CPU, GPU, and Hybrid approaches.

Figure 33 shows the weak scaling efficiency. From this plot we can see that there are two modes. At node levels 2, 8, and 32 we can see a downward trend that drops from 100% efficiency to around 93% efficiency for all approaches. However, at node levels 1, 4, and 16 we can see a significantly better trend where the data only drops from 100% to 99.5% for all approaches. This is an interesting performance characteristic that warrants future study, but in either case shows very strong weak scaling efficiency for this problem.

Figure 34 shows the speedup for Hybrid and GPU only approaches over the CPU only approach. This shows that the GPUs are very effective at providing performance for this problem, giving greater than 7× speedup for all scales. In addition, the Hybrid approach shows performance benefits at all scales over the GPU only approach with most speedup values at or above 8×. This data shows

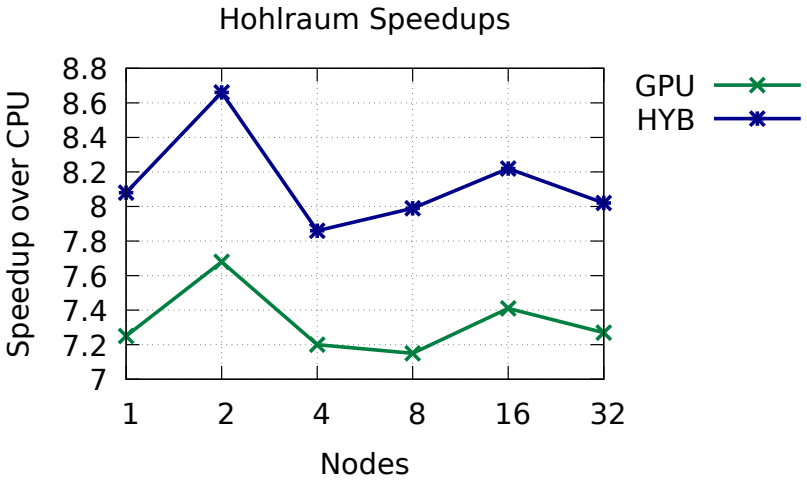
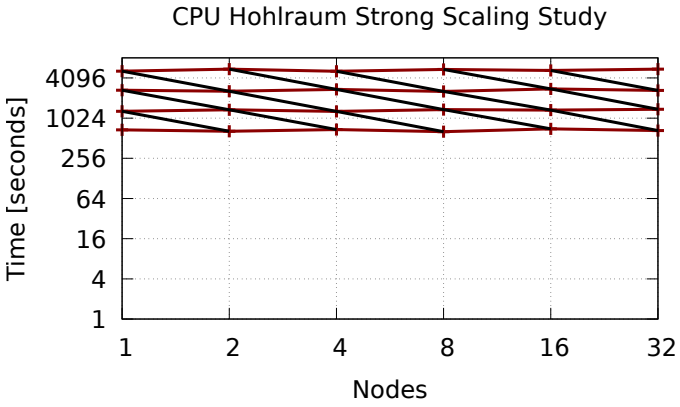


Figure 34. Speedup of GPU and Hybrid approached over CPU only approach

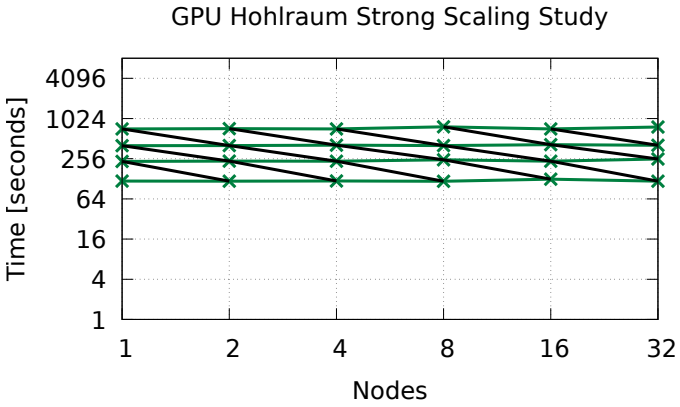
that the Hybrid approach is very effective at providing speedups for this problem and continues to provide speedup even up to 32 nodes.

Figure 35 shows the strong scaling ability of each approach through four weak scaling studies. In this plot we can see that the CPU results show nearly perfect weak scaling at all levels, and the strong scaling lines appear to follow the correct pattern, indicating good strong scaling as well. The GPU results show good weak scaling but with a slight upward trend which means that while this is still strong scaling well it is less than perfect. Finally, the Hybrid results show nearly identical performance characteristics as the GPU results, showing that the addition of the CPUs did not cause a significant loss in strong or weak scaling performance.

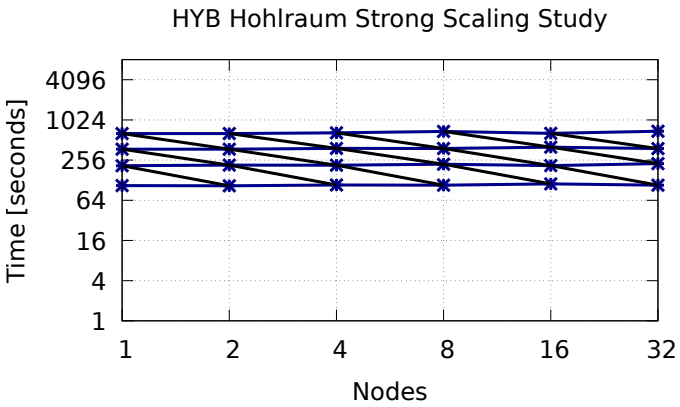
Figure 36 shows the strong scaling efficiencies of each strong scaling line in the previous weak/strong plots. This data indicates that the previous analysis was correct. The CPU results show excellent strong scaling staying at nearly 100% load balance at all scales. The GPU and Hybrid results show that at most scales the strong scaling drops to around 80% efficient, which makes sense for the GPU



(a) CPU approach

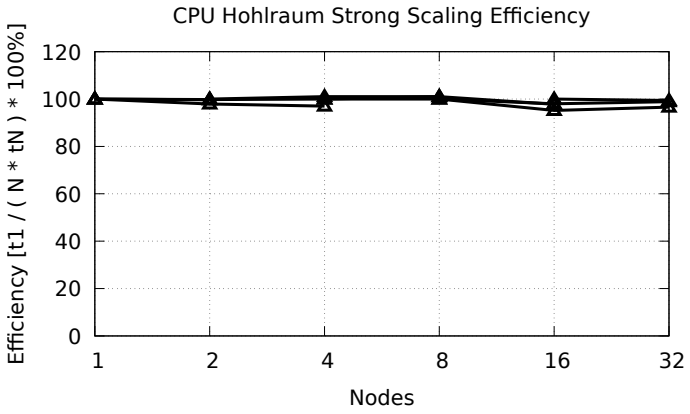


(b) GPU approach

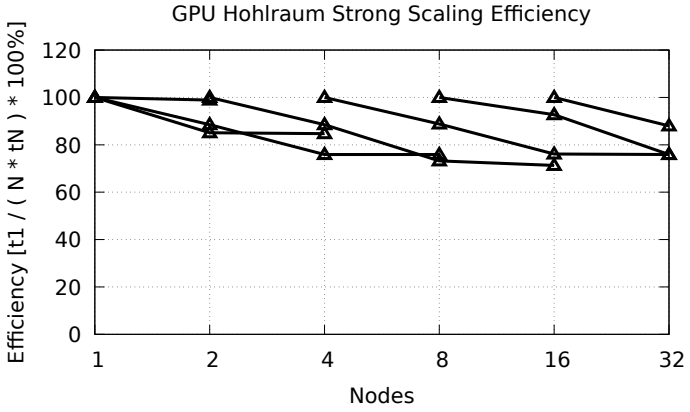


(c) Hybrid approach

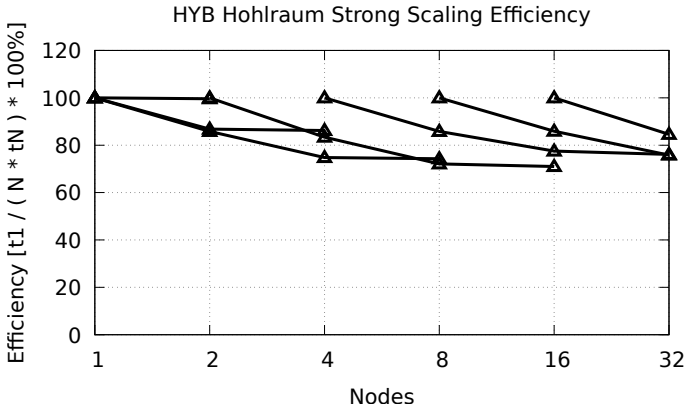
Figure 35. A partial strong scaling study of the Hohlraum problem using (a) CPU, (b) GPU, and (c) Hybrid approaches.



(a) CPU approach



(b) GPU approach



(c) Hybrid approach

Figure 36. Efficiency of each strong scaling line from each of the studies done for (a) CPU, (b) GPU, and (c) Hybrid cases.

platforms. Since GPUs need a large amount of work to remain saturated and operating efficiently, the GPU architecture will strong scale poorly once the work saturation limit is reached — i.e. once the amount of work per GPU drops below the limit that keeps the GPUs fully saturated they will not strong scale well.

7.3.3 Mercury Godiva In Water Analysis. The Godiva in water problem in Mercury is described in detail in subsubsection 7.2.4.3 (Godiva In Water). The primary reason to include this problem in this study is that the collision physics routines in Mercury are far more complex that in Imp, requiring calls into a nuclear data library, a deeper call stack, and more options for divergence. This problem provides a good example of the factors that make Monte Carlo transport problems difficult to run on GPUs.

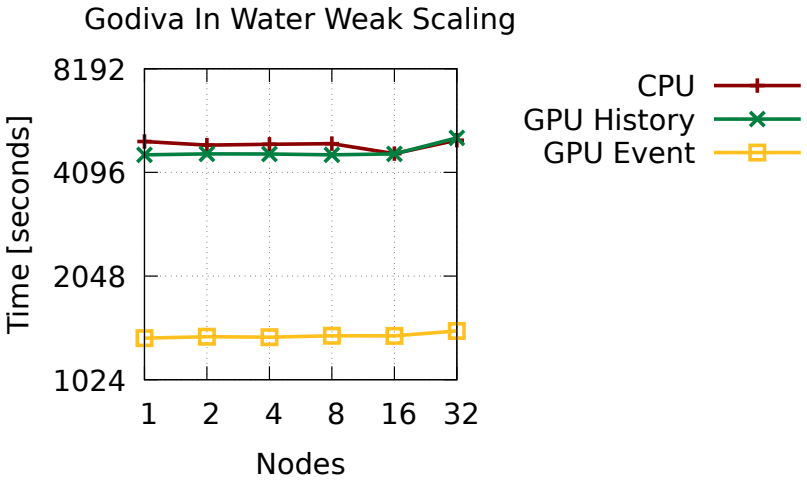


Figure 37. Weak scaling study for Godiva in water problem for CPU, GPU history-based, and GPU event-based approaches on up to 32 nodes.

The first study to understand performance is to evaluate the weak scaling ability of Mercury on CPUs and GPUs, as show in Figure 37. In addition to the history-based approach, included are results for our preliminary implementation of an event-based approach. The first observation is that the GPU history

based approach is almost exactly the same speed as the CPU only approach. Additionally, we can see there is performance benefit from an event-based approach on this problem. Furthermore, all variations of this problem are weak scaling nearly perfectly up to 32 nodes.

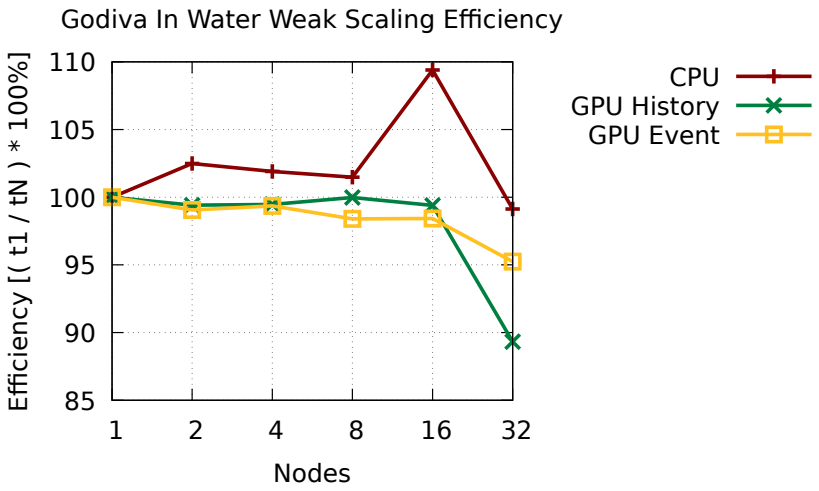


Figure 38. Weak scaling efficiencies for Godiva in water problem for CPU, GPU history-based, and GPU event-based approaches on up to 32 nodes.

To better understand the weak scaling results we evaluate the weak scaling efficiencies. Figure 38 gives the weak scaling efficiencies corresponding to the weak scaling study. From this data we can see that the CPUs are scaling at better than perfect weak scaling. Additionally, the history-based and event-based GPU approaches are weak scaling perfectly until 32 nodes. At 32 nodes the history-based approach drops to ∼90% efficiency while the event-based approach drops to ∼95% efficiency.

Figure 39 shows the speedups for our GPU approaches over the CPU approach. We can see that our history-based approach sits right along the 1× speedup while the event-based approach is closer to 3.5× speedup at all scales.

7.3.4 Imp Mercury Comparisons.

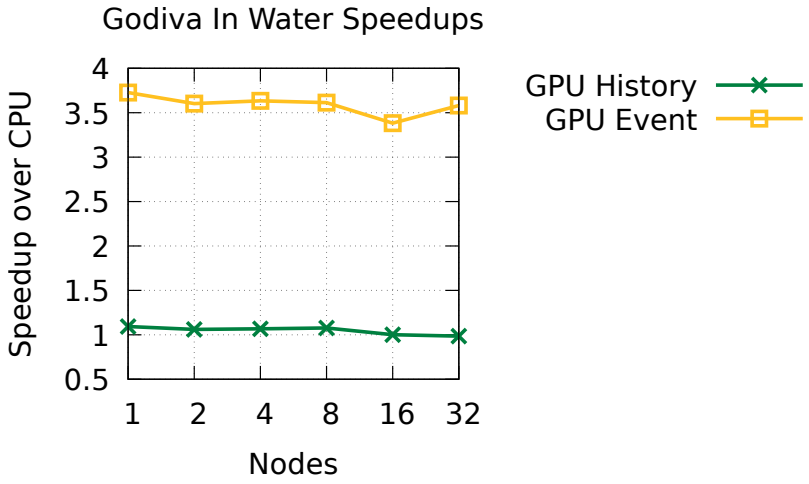


Figure 39. Speedup of GPU history-based and GPU event-based approaches over CPU only approach.

Evaluating the differences in speedup between all of the results presented in this chapter provides insight into the problems faced by Monte Carlo transport applications. Focusing on the Imp GPU speedup results we can see that for Crooked Pipe GPUs produce a speedup between 3-4.6×, while the Hohlraum problem has speedups around 7.3×. The difference between these two problems is that the Hohlraum problem is mostly photon collision physics while the Crooked Pipe problem is mostly particle streaming. Comparing this result to the history-based GPU speedup result from Mercury, ∼1×, we can see that the complexity of the Mercury collision physics is not being handled well by the GPUs since it is less efficient than both of the Imp results. Only when we added an event-based method did we start to see some performance benefits, ∼3.5×.

For all results in all problems, we did not see any significant performance degradation due to MPI scaling. More scaling is necessary to ensure that this holds true at massive scale, but on a reasonably large number of GPUs we showed we could maintain performance and weak and strong scale effectively.

7.4 Conclusion

In this chapter we analyzed the performance of Monte Carlo transport applications on GPUs and at scale. From this data we showed good performance and excellent scalability for fully featured problems run on 32 nodes of RZAnsel. We showed that by combining the ideas presented in this dissertation into a fully featured application we could achieve performance and maintain excellent scalability.

For the simpler photon collision physics problems in Imp, we showed that a history-based approach is viable and able to achieve over a 8.5× speedup, node to node, versus using CPU only. In addition, for the complex collision physics problems in Mercury we showed that an event-based approach was able to achieve a 3.5× speedup, node to node. In addition to speedup, we showed that almost perfect weak-scaling and excellent strong scaling results are maintained when running on GPUs.

CHAPTER VIII

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

This dissertation answers the research question:

What changes to Monte Carlo particle transport algorithms will enable effective utilization of many-core architectures?

Answering this question required answering a series of shorter questions which together provide a complete picture.

Chapter III (Tracking Algorithms) addresses the research question: What tracking algorithms are best suited for portable performance of Monte Carlo transport on modern many-core systems? This chapter shows a modernized event-based algorithm which is implemented and tested against a GPU optimized version of the history-based algorithm. The result of this study was to find that in a small scale application both methods are valid and performant on the GPU. In fact, even at larger scale both methods have now been shown to work, but as the size of an application scales up, the more challenging it becomes to get performance out of a history-based approach on GPUs. In all, our primary finding is that an event-based approach is the most suitable for many-core architectures. Finally, this work also addresses portability by introducing the Thrust parallel abstraction and showing the potential for enabling both GPU performance as well as CPU threaded performance.

Chapter IV (Data Race Management: Threading Models) and Chapter V (Data Race Management: Output Tally Data) together answer the research question: What is the best way to manage data-races and the memory needs of many-core platforms? Chapter IV looks at data management through developing

a new threading model and defining the differences between the threading models. This work showed that our Thin-Threads threading model — one that focuses on reducing redundant information and streamlines the data a single thread needs — is a performant solution for GPUs and CPUs alike. Chapter V looks at data management through developing methods for handling output tally data, namely variable replication with atomics. We introduce the idea of variable replication and show that through a combination of variable replication and atomics we can implement the Thin-Threads model without losing performance due to atomic operations colliding on memory accesses. We also studied the performance of atomics on GPUs to understand the impact different types of memory access patterns have on the performance of an atomic operation. Together we show a concrete methodology for handling the race-conditions and data needs of many-core machines. This work also shows results on CPU based platforms, showing that the change in threading model did not negatively impact performance on this architecture, and maintained good scalable efficiency at over 24 thousand nodes on LLNL's Vulcan supercomputer.

Chapter VI (Heterogeneous Architecture Utilization) answers the question: Is it worthwhile to fully utilize heterogenous node architectures? In this chapter we introduce a new dynamic replication load balancing scheme and introduce the idea that CPUs can be used for compute alongside the GPUs on modern systems. We show in practice that we can gain 20% additional performance by including the CPUs in the calculation. This work shows that new load balancing algorithms are needed to utilize a heterogeneous node architecture if all processing elements are going to be used for compute. This work handles portability by introducing a method that works for load balancing both homogeneous or heterogeneous systems.

In addition, it makes no assumptions about the hardware performance and instead relies on collecting data during a run or user provided input to make its decisions for how to assign work to processors.

Chapter VII (Performance at Scale) answers the question: How does many-core focused algorithm development impact performance concerns as we scale up MPI resources? In this chapter we analyze the scaling performance of Imp and Mercury on up to 32 nodes. In all problems that we ran, both codes showed excellent scaling performance. In addition, we demonstrated performance of up to 8.5× a CPU only approach in Imp and 3.5× a CPU only approach in Mercury. In this analysis we demonstrated that a history-based approach works well for Imp and an event-based approach works better for Mercury. All together these results demonstrated a successful approach to Monte Carlo particle transport on GPUs.

The primary research question has been answered by taking the combination of all the presented works. All together and exemplified in the studies given in Chapter VII, are described: an effective transport algorithm, a concrete threading model, new data management techniques, and a new method for fully utilizing heterogeneous node architectures. In addition, all of these solutions were provided in a single source code base that works for CPU and GPU systems without the need for recompilation, or significant amounts of architecture-specific coding.

All of the work in this dissertation followed a development strategy that provided a space for developing research concepts quickly and then extending them into a full implementation in a large production application. The proxy apps, Quicksilver and ALPSMC, were used for prototyping and testing initial research concepts. The production applications, Mercury and Imp, were the locations for full implementations of these concepts. The use of proxy apps has enabled faster

prototyping of different approaches as well as provided an avenue for hardware vendor interactions. Additionally, access to the production applications has enabled testing at scale and validation of previous experiments in real world settings. This process was necessary since a requirement and goal of my research was the eventual, full implementation of these ideas into a full scale production application in order to support the LLNL mission.

In addition to the specific benefits to Monte Carlo particle transport, there are many aspects of the works presented in this dissertation that can be generalized to a larger community. Many large multi-physics applications face similar struggles when developing for many-core applications and will benefit from this knowledge. Three primary take aways for any application considering many-core environments include:

1. In large divergent kernels, divergence is not the primary problem if the algorithm is memory-latency bound. As long as there is enough parallel work to be done, context switching while waiting for memory will hide much of the divergence. Namely, divergence is not the primary performance factor and efforts to improve memory access times generally improve performance the most. This is exemplified in the study looking at history-based and event-based tracking algorithms in chapter III; the performance benefit from switching to event-based came only after many memory related optimizations were added. In addition, in the final study with Mercury it is the case that splitting the kernel from history-based to event-based provided performance primarily because of the large memory requirement of the more complex neutron physics, meaning splitting this functionality out from the rest of the work was able to provide greater benefit.

2. The performance of atomics on GPUs is not a problem if the memory system is already heavily affected by the number or size of memory reads and writes. Additionally, the concept of variable replication can be applied to any algorithm that can benefit from the tradeoff of memory for performance.

3. Heterogeneous load balancing is possible and can be done efficiently. The algorithm presented in Chapter VI will work with any workload and distribution of processor speeds.

8.2 Future Work

While this dissertation provides a path forward for Monte Carlo transport to efficiently utilize many-core architectures, additional directions may yield even more performance. In this section we will address these areas and provide initial ideas and thoughts for future work.

8.2.1 Algorithm Development. In this dissertation we addressed the history- versus event-based algorithm debate. The difficulty however, is that both solutions are doable, and can be made performant under the right conditions. To take this to the next level, a much more detailed analysis of the possible optimizations for each approach could be considered.

One such optimization example is to better understand the possibility for simplifying the history-based approach with macros, templating, or other compile time optimizations, which could significantly impact the optimizations a compiler could make as well as changing the number of registers needed for a kernel dramatically. Extending this concept, if a single large kernel could separate the complex/divergent/expensive code paths into compile time known paths, then a set of simpler kernels could be devised, each of which is faster than the single large kernel. If a particle falls into the category that allows for it to take this optimized

fast path, it will, and if it cannot, a slower less optimized kernel could be used for a subset of those particles which were not "fast" particles. A scheme such as this has been the topic of discussion in Monte Carlo transport community meetings, but to my knowledge has not been fully developed outside of simple, proof of concept tests. A simple extension would be to do the same thing for each of the event-based kernels.

Subsequently, if this approach is viable there are probably many more optimizations that can be pursued that might offer more performance for either history-based or event-based. In all, reducing kernel complexity is a very promising area that is worthy of exploration.

8.2.2 Memory Management. In this dissertation we introduced the idea of variable replication. We showed that this concept works, and works well. It has the added benefit of fitting in nicely with the work done so far on threading models, and with the way production codes already handle tally data. Other tally data schemes exist however, and it would be a goal for future research to look into these other schemes and see if any offer added performance for GPU platforms. Many of these other schemes are avoided due to added complexity but offer a possible use of a heterogeneous systems resources in interesting ways.

One concept that could be further explored is the idea of a tally server designed for GPU systems. In this concept tallies are processed by a separate entity which manages atomics or reduction. Each thread, instead of writing a tally directly, provides its data to the tally server. This data can be provided directly, or as a series of replays, where each thread simply keeps a list of tallies it needs to perform and handles those off to the tally server at certain synchronization points. In this way threads never have to worry about keeping the full state of the tally

data, and do not contend with each other for shared memory resources. There are obvious complications and limitations to this concept, but there are also interesting use cases that make this concept worthy of investigation. For example, if the CPUs on a system run a tally server while the GPUs perform compute, then the CPUs can communicate reductions and manage data while the GPUS focus on tracking particles. This provides a use for CPUs which are often left idle on heterogeneous systems. Successfully implementing this concept would require CPUs to collect data from GPU memory quickly enough that the GPUs could keep processing without stalling. In addition, the CPUs could handle most of the reductions ahead of time, ensuring that the finalization phase of a cycle is performed quickly.

8.2.3 Performance and Heterogeneous Architectures. The work we presented in Chapter VI demonstrated the value of a heterogeneous computational model, but there are still more areas to explore in this space. For example, step 3 of the dynamic replication algorithm is to develop a communication map for processors. There are a large number of possible mappings that can be made and optimizing this can lead to increased performance and greater scalability. In addition, complementary schemes could be implemented, such as dividing the work between two problems, one homogeneous GPU problem with some percent of the work, and a homogeneous CPU problem with the remaining percent of the work. The two subproblems could then be run together and their results could be combined. This and other concepts could be tested and might provide a simpler way to achieve hybrid performance without requiring CPUs and GPUs to work together on compute at once.

Studying all of this work at a significantly larger scale is also important. It would be very useful to know if any of the presented ideas or concepts break down

at larger and larger node counts. For example, the Mercury simulation code scaled well until it was run on over one million processors on Sequoia. At that point new load balancing algorithms were required that did not scale poorly with number of processors. While this dissertation's scaling study mitigates some of the risk of scaling inefficiency, considering tens of thousands of GPUs would fully answer the question.

REFERENCES CITED

About trinity. (2015). Retrieved 1-28-2021, from `http://www.lanl.gov/projects/trinity/about.php`

Alerstam, E., Svensson, T., & Andersson-Engels, S. (2008). Parallel computing with graphics processing units for high-speed monte carlo simulation of photon migration. *Journal of biomedical optics*, *13*(6), 060504–060504.

Alme, H. J., Rodrigue, G. H., & Zimmerman, G. B. (2001). Domain decomposition models for parallel monte carlo transport. *The Journal of Supercomputing*, *18*(1), 5–23.

Antithetic variates. (2021). Wikimedia Foundation. Retrieved from `https://en.wikipedia.org/wiki/Antithetic_variates`

Ayguadé, E., Badia, R. M., Bellens, P., Cabrera, D., Duran, A., Ferrer, R., ... others (2010). Extending openmp to survive the heterogeneous multi-core era. *International Journal of Parallel Programming*, *38*(5-6), 440–459.

Badal, A., & Badano, A. (2009). Accelerating monte carlo simulations of photon transport in a voxelized geometry using a massively parallel graphics processing unit. *Medical physics*, *36*(11), 4878–4880.

Bell, N., & Hoberock, J. (2011). Thrust: A productivity-oriented library for cuda. *GPU computing gems Jade edition*, *2*, 359–371.

Bergmann, R. M. (2014). The development of warp-a framework for continuous energy monte carlo neutron transport in general 3d geometries on gpus.

Bleile, R., Brantley, P., Dawson, S., McKinley, M. S., O'Brien, M., Richards, D., & Childs, H. (2019, July). Thin-threads: An approach for history-based monte carlo on gpus. , 273–280.

Bleile, R., Brantley, P., Dawson, S., O'Brien, M., & Childs, H. (2016). Investigation of portable event-based monte carlo transport using the nvidia thrust library. *Transactions of the American Nuclear Society(ANS)*(114), 941-944.

Bleile, R., Brantley, P., O'Brien, M., & Childs, H. (2016). Algorithmic improvements for portable event-based monte carlo transport using the nvidia thrust library. *Transactions of the American Nuclear Society(ANS)*(115), 535-538.

Bleile, R., Brantley, P., O'Brien, M., & Childs, H. (2021). Scalability and performance of monte carlo transport on gpus.

Bleile, R., Brantley, P., O'Brien, M., & Childs, H. ((in-submission) 2021, June). A dynamic replication approach for monte carlo transport on heterogeneous architectures.

Blelloch, G. E. (1990). *Vector models for data-parallel computing* (Vol. 356). MIT press Cambridge.

Bobrowicz, F., Lynch, J., Fisher, K., & Tabor, J. (1984). Vectorized monte carlo photon transport. *Parallel Computing*, *1*(3), 295–305.

Bosilca, G., Bouteiller, A., Herault, T., Lemarinier, P., Saengpatsa, N. O., Tomov, S., & Dongarra, J. J. (2011). Performance portability of a gpu enabled factorization with the dague framework. , 395–402.

Brantley, P., Dawson, S., McKinley, M., O'Brien, M., Stevens, D., Beck, B., ... Hall, J. (2013). Recent advances in the mercury monte carlo particle transport code. , 5–9.

Brantley, P., et al. (August, 2019). A new implicit monte carlo thermal photon transport capability developed using shared monte carlo infrastructure. , 25-29.

Brantley, P. S. (2011). A benchmark comparison of monte carlo particle transport algorithms for binary stochastic mixtures. *Journal of Quantitative Spectroscopy and Radiative Transfer*, *112*, 599–618.

Brown, F. B. (2011). Recent advances and future prospects for monte carlo. *Progress in nuclear science and technology*, *2*, 1–4.

Brown, F. B. (2014). New hash-based energy lookup algorithm for monte carlo codes. *Trans. Am. Nucl. Soc*, *111*, 659–662.

Brown, F. B., & Martin, W. R. (1984). Monte carlo methods for radiation transport analysis on vector computers. *Progress in Nuclear Energy*, *14*, 269–299.

Brunner, T. A., & Brantley, P. S. (2009). An efficient, robust, domain-decomposition algorithm for particle monte carlo. *Journal of Computational Physics*, *228*(10), 3882–3890.

Brunner, T. A., et al. (2006). Comparison of four parallel algorithms for domain decomposed implicit monte carlo. *Journal of Computational Physics*, *212*(2), 527–539.

Burns, P. J., Christon, M., Schweitzer, R., Lubeck, O. M., & Wasserman, H. J. (1989). Vectorization on monte carlo particle transport: an architectural study using the lanl benchmark "gamteb". , 10–20.

Chamberlain, B. (2013, may). *Chapel: Productive parallel programming.* (http://www.cray.com/blog/chapel-productive-parallel-programming/)

Chamberlain, B. L., Callahan, D., & Zima, H. P. (2007). Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, *21*(3), 291–312.

Clarke, L., Glendinning, I., & Hempel, R. (1994). The mpi message passing interface standard. , 213–218.

Co-design at lawrence livermore national lab: Quicksilver. (2017). Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States). Retrieved from `https://codesign.llnl.gov/quicksilver.php` (Accessed: 2017-07-07)

Control variates. (2021). Wikimedia Foundation. Retrieved from `https://en.wikipedia.org/wiki/Control_variates`

Cuda. (2014). (http://www.nvidia.com/object/cuda_home_new.html)

Cullen, D. E., Clouse, C. J., Procassini, R., & Little, R. C. (2003). *Static and dynamic criticality: are they different?* (Tech. Rep.). Lawrence Livermore National Lab., Livermore, CA (US). (Report: UCRL-TR-201506)

Després, P., Rinkel, J., Hasegawa, B. H., & Prevrhal, S. (2008). Stream processors: a new platform for monte carlo calculations. , *102*(1), 012007.

Ding, A., Liu, T., Liang, C., Ji, W., Shephard, M. S., Xu, X. G., & Brown, F. B. (2011). Evaluation of speedup of monte carlo calculations of two simple reactor physics problems coded for the gpu/cuda environment.

Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., & Dongarra, J. (2012). From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, *38*(8), 391–407.

Du, X., Liu, T., Ji, W., Xu, X., & Brown, F. (2013). *Evaluation of vectorized monte carlo algorithms on gpus for a neutron eigenvalue problem* (Tech. Rep.). American Nuclear Society, 555 North Kensington Avenue, La Grange Park, IL 60526 (United States).

Eckhardt, R. (1987). Stan ulam, john von neumann, and the monte carlo method. *Los Alamos Science*, *15*(131-136), 30, LA-UR-88-9068.

Edwards, H. C., & Sunderland, D. (2012). Kokkos array performance-portable manycore programming model. , 1–10.

Edwards, H. C., Sunderland, D., Porter, V., Amsler, C., & Mish, S. (2012). Manycore performance-portability: Kokkos multidimensional array library. *Scientific Programming*, *20*(2), 89–114.

Edwards, H. C., Trott, C. R., & Sunderland, D. (2014). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, *74*(12), 3202–3216.

Ellis, J. A., et al. (2019). Optimization of processor allocation for domain decomposed monte carlo calculations. *Parallel Computing*, *87*, 77–86.

El-Rewini, H., & Abd-El-Barr, M. (2005). *Advanced computer architecture and parallel processing* (Vol. 42). John Wiley & Sons.

Fact sheet: Collaboration of oak ridge, argonne, and livermore (coral). (2014). Retrieved from `https://www.energy.gov/downloads/fact-sheet-collaboration-oak-ridge-argonne-and-livermore-coral`

Floating point and ieee 754. (2015, Sep). NVIDIA Coorperation. Retrieved from `http://docs.nvidia.com/cuda/floating-point/#axzz4k4zi4wrv`

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, *23*(1), 5–48.

Gong, C., Liu, J., Yang, B., Deng, L., Li, G., Li, X., ... Gong, Z. (2011). Accelerating mcnp-based monte carlo simulations for neutron transport on gpu. *International Journal of Radiation Oncology* Biology* Physics*, *81*(2), S157–S158.

Graziani, F., & LeBlanc, J. (2000). *The crooked pipe test problem* (Tech. Rep.). Lawrence Livermore National Lab., Livermore, CA (US). (Report: UCRL-MI-143393)

Greenman, G., O'Brien, M., Procassini, R., & Joy, K. (2009). Enhancements to the combinatorial geometry particle tracker in the mercury monte carlo transport code: Embedded meshes and domain decomposition.

Hamilton, S. P., & Evans, T. M. (2019). Continuous-energy monte carlo neutron transport on gpus in the shift code. *Annals of Nuclear Energy*, *128*, 236–247.

Hamilton, S. P., Slattery, S. R., & Evans, T. M. (2018). Multigroup monte carlo on gpus: Comparison of history-and event-based algorithms. *Annals of Nuclear Energy*, *113*, 506–518.

Heimlich, A., Mol, A. C., & Pereira, C. M. (2009). Gpu-based high performance monte carlo simulation in neutron transport.

Hoberock, J., & Bell, N. (2010). Thrust: A parallel template library. *Online at http://thrust. googlecode. com*, *42*, 43.

Horelik, N., Siegel, A., Forget, B., & Smith, K. (2014). Monte carlo domain decomposition for robust nuclear reactor analysis. *Parallel Computing*, *40*(10), 646–660.

Hornung, R., Keasler, J., Kunen, A., Jones, H., & Beckingsale, D. (2016). *Raja-llnl hpc architecture portability encapsulation layer version 1.0* (Tech. Rep.). Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States).

Hornung, R., Keasler, J., et al. (2014). The raja portability layer: overview and status. *Lawrence Livermore National Laboratory, Livermore, USA*.

Iwabuchi, H. (2015). Efficient monte carlo methods for radiative transfer modeling. *Journal of the atmospheric sciences*, *72*(9).

Jahnke, L., Fleckenstein, J., Wenz, F., & Hesser, J. (2012). Gmc: a gpu implementation of a monte carlo dose calculation based on geant4. *Physics in medicine and biology*, *57*(5), 1217.

Jia, X., Gu, X., Graves, Y. J., Folkerts, M., & Jiang, S. B. (2011). Gpu-based fast monte carlo simulation for radiotherapy dose calculation. *Physics in medicine and biology*, *56*(22), 7017.

Jia, X., Gu, X., Sempau, J., Choi, D., Majumdar, A., & Jiang, S. B. (2010). Development of a gpu-based monte carlo dose calculation code for coupled electron–photon transport. *Physics in medicine and biology*, *55*(11), 3077.

Jia, X., Schümann, J., Paganetti, H., & Jiang, S. B. (2012). Gpu-based fast monte carlo dose calculation for proton therapy. *Physics in medicine and biology*, *57*(23), 7783.

Kahn, H., & Marshall, A. W. (1953). Methods of reducing sample size in monte carlo computations. *Journal of the Operations Research Society of America*, *1*(5), 263–278.

Kokkos. (2021). Retrieved from `https://github.com/kokkos/kokkos` (v. 3.3.1)

Larsen, M., Labasan, S., Navrátil, P. A., Meredith, J. S., & Childs, H. (2015). Volume rendering via data-parallel primitives. , 53–62.

Larsen, M., Meredith, J. S., Navrátil, P. A., & Childs, H. (2015). Ray tracing within a data parallel framework. , 279–286.

Lee, S., & Eigenmann, R. (2010). Openmpc: Extended openmp programming and tuning for gpus. , 1–11.

Lee, S., Min, S.-J., & Eigenmann, R. (2009). Openmp to gpgpu: a compiler framework for automatic translation and optimization. *ACM Sigplan Notices*, *44*(4), 101–110.

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., ... others (2010). Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *ACM SIGARCH Computer Architecture News*, *38*(3), 451–460.

Leppänen, J. (2009). Two practical methods for unionized energy grid construction in continuous-energy monte carlo neutron transport calculation. *Annals of Nuclear Energy*, *36*(7), 878–885.

Lewis, E. E., & Miller, W. F., Jr. (1993). *Computational methods of neutron transport*. 555 N. Kensington Avenue La Grange Park, Illinois 60525 USA: American Nuclear Society, Inc.

Liu, T., Du, X., Ji, W., Xu, X. G., & Brown, F. B. (2014). A comparative study of history-based versus vectorized monte carlo methods in the gpu/cuda environment for a simple neutron eigenvalue problem. , 04206.

Liu, T., Xu, X. G., & Carothers, C. D. (2015). Comparison of two accelerators for monte carlo radiation transport calculations, nvidia tesla m2090 gpu and intel xeon phi 5110p coprocessor: A case study for x-ray ct imaging dose calculation. *Annals of Nuclear Energy*, *82*, 230–239.

Livermore computing center high performance computing: Rzansel. (2020). Retrieved from `https://hpc.llnl.gov/hardware/platforms/rzansel` (Accessed: 2020-12-09)

Lo, L.-t., Sewell, C., & Ahrens, J. P. (2012). Piston: A portable cross-platform framework for data-parallel visualization operators. , 11–20.

Lund, A., & Siegel, A. (2015). Using fractional cascading to accelerate cross section lookups in monte carlo neutron transport calculations. In *Ans m&c 2015*. LaGrange Park, IL.

Lux, I., & Koblinger, L. (1991). *Monte carlo particle transport methods: Neutron and photon calculations*. 2000 Corporate Blvd., Boca Raton, Florida 33431: CRC Press, Inc.

Majumdar, A. (2000). Parallel performance study of monte carlo photon transport code on shared-, distributed-, and distributed-shared-memory architectures. , 93–99.

Martin, W. R. (1989). Successful vectorization-reactor physics monte carlo code. *Computer Physics Communications*, *57*(1-3), 68–77.

Martin, W. R., Nowak, P. F., & Rathkopf, J. A. (1986). Monte carlo photon transport on a vector supercomputer. *IBM Journal of Research and Development*, *30*(2), 193–202.

McKinley, M., & Beck, B. (2015). Implementation of the generalized interaction data interface (gidi) in the mercury monte carlo code.

Melnik-Melnikov, P., & Dekhtyaruk, E. (2000). Rare events probabilities estimation by "russian roulette and splitting" simulation technique. *Probabilistic engineering mechanics*, *15*(2), 125–129.

Mercury. (2019). Retrieved from `https://wci.llnl.gov/simulation/computer-codes/mercury` (Accessed: 2019-12-12)

Meredith, J., Ahern, S., Pugmire, D., & Sisneros, R. (2016). *Eavl.* (http://ft.ornl.gov/eavl/index.html)

Moreland, K., Ayachit, U., Geveci, B., & Ma, K.-L. (2011). Dax toolkit: A proposed framework for data analysis and visualization at extreme scale. , 97–104.

Moreland, K., Ayachit, U., Geveci, B., & Ma, K.-L. (2016). Dax: Data analysis at extreme.

Moreland, K., Larsen, M., & Childs, H. (2015). Visualization for exascale: Portable performance is critical. *Supercomputing frontiers and innovations*, *2*(3), 67–75.

Moreland, K., Sewell, C., Usher, W., Lo, L., Meredith, J., Pugmire, D., ... Geveci, B. (2016, May/June). VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications (CG&A)*, *36*(3), 48-58.

Nelson, A. G. (2009). *Monte carlo methods for neutron transport on graphics processing units using cuda.* The Pennsylvania State University: M.S. Thesis.

NVIDIA. (2015). *Cuda c programming guide.* (version 7.5)

O'Brien, M. (2007). Dynamic load balancing of parallel monte carlo transport calculations via spatial redecomposition. , 16–19.

O'Brien, M., & Brantley, P. (2015). Particle communication and domain neighbor coupling: Scalable domain decomposed algorithms for monte carlo particle transport. *Lawrence Livermore National Laboratory (LLNL), Livermore*.

O'Brien, M., Joy, K., Procassini, R., & Greenman, G. (2009). Domain decomposition of a constructive solid geometry monte carlo transport code.

O'Brien, M., et al. (2019). Hybrid cpu-gpu load balancing for monte carlo particle transport.

O'Brien, M., Taylor, J., & Procassini, R. (2005). Dynamic load balancing of parallel monte carlo transport calculations. *The Monte Carlo Method: Versatility Unbounded In A Dynamic Computing World*, 17–21.

O'Brien, M. J., Brantley, P. S., & Joy, K. I. (2013). Scalable load balancing for massively parallel distributed monte carlo particle transport. , *45*(13), 647-658.

Openacc. (2018). Retrieved from `http://www.openacc.org/`

Openmp. (2018). Retrieved from `http://www.openmp.org/`

Optix programming guide. (2018). NVIDIA Coorperation. Retrieved from `http://docs.nvidia.com/gameworks/content/gameworkslibrary/optix/optix_programming_guide.htm`

Ozog, D., Malony, A. D., & Siegel, A. R. (2015). A performance analysis of simd algorithms for monte carlo simulations of nuclear reactor cores. , 733–742.

Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., ... others (2010). Optix: a general purpose ray tracing engine. , *29*(4), 66.

Pharr, M., & Mark, W. R. (2012). ispc: A spmd compiler for high-performance cpu programming. *Innovative Parallel Computing (InPar)*, 1–13.

Piston, a portable cross-platform framework for data-parallel visualization operators. (2016). (http://viz.lanl.gov/projects/PISTON.html)

Procassini, R., O'Brien, M., & Taylor, J. (2005). Load balancing of parallel monte carlo transport calculations. *Mathematics and Computation, Supercomputing, Reactor Physics and Nuclear and Biological Applications, Palais des Papes, Avignon, Fra.*

Pvm parallel virtual machines. (2011, dec). (http://www.csm.ornl.gov/pvm/pvm_home.html)

Quicksilver. a proxy app for the monte carlo transport code, mercury. llnl-code-684037. (2017). Retrieved from `https://github.com/LLNL/Quicksilver` (Version: 83ade89)

Rahaman, R., Medina, D., Lund, A., Tramm, J., Warburton, T., & Siegel, A. (2015). Portability and performance of nuclear reactor simulations on many-core architectures. , 42–47.

Raja. (2021). Retrieved from `https://github.com/LLNL/RAJA` (v0.13.0)

Ray tracing (graphics). (2021). Wikimedia Foundation. Retrieved from `https://en.wikipedia.org/wiki/Ray_tracing_(graphics)`

Ren, N., Liang, J., Qu, X., Li, J., Lu, B., & Tian, J. (2010). Gpu-based monte carlo simulation for light propagation in complex heterogeneous tissues. *Optics express*, *18*(7), 6811–6823.

Richards, D. F., Bleile, R. C., Brantley, P. S., Dawson, S. A., McKinley, M. S., & O'Brien, M. J. (2017). Quicksilver: A proxy app for the monte carlo transport code mercury. , 866–873.

Romano, P. K. (2013). *Parallel algorithms for monte carlo particle transport simulation on exascale computing architectures* (Unpublished doctoral dissertation). Massachusetts Institute of Technology.

Romano, P. K., & Forget, B. (2013). The openmc monte carlo particle transport code. *Annals of Nuclear Energy*, *51*, 274–281.

Romano, P. K., Horelik, N. E., Herman, B. R., Nelson, A. G., Forget, B., & Smith, K. (2015). Openmc: A state-of-the-art monte carlo code for research and development. *Annals of Nuclear Energy*, *82*, 90–97.

Russell, R. M. (1978). The cray-1 computer system. *Communications of the ACM*, *21*(1), 63–72.

Scudiero, A. (2016). *personal communication.*

Sidelnik, A., Maleki, S., Chamberlain, B. L., Garzar'n, M. J., & Padua, D. (2012). Performance portability with the chapel language. , 582–594.

Siegel, A. R., Smith, K., Romano, P. K., Forget, B., & Felker, K. G. (2014). Multi-core performance studies of a monte carlo neutron transport code. *International Journal of High Performance Computing Applications*, *28*(1), 87–96.

Snavely, A., Carter, L., Boisseau, J., Majumdar, A., Gatlin, K. S., Mitchell, N., ... Koblenz, B. (1998). Multi-processor performance on the tera mta. , 1–8.

Stratified sampling. (2021). Wikimedia Foundation. Retrieved from https://en.wikipedia.org/wiki/Stratified_sampling

Su, L., Du, X., Liu, T., & Xu, X. (2013). *Monte carlo electron-photon transport using gpus as an accelerator: Results for a water-aluminum-water phantom* (Tech. Rep.). American Nuclear Society, 555 North Kensington Avenue, La Grange Park, IL 60526 (United States).

Thomas, J. (March 2020). *Llnl and hpe to partner with amd on el capitan, projected as world's fastest supercomputer.* Retrieved 01-28-2021, from https://www.llnl.gov/news/llnl-and-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer

Thrust. (2014). (https://developer.nvidia.com/Thrust)

Thrust - parallel algorithms library. (2018). Retrieved from http://thrust.github.io/

Tickner, J. (2010). Monte carlo simulation of x-ray and gamma-ray photon transport on a graphics-processing unit. *Computer Physics Communications*, *181*(11), 1821–1832.

Top 500 list. (2019). Retrieved from https://www.top500.org/lists/2019/11/ (Accessed: 2019-12-12)

Tramm, J. R., Siegel, A. R., Islam, T., & Schulz, M. (2014). Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR).*

Variance reduction. (2021). Wikimedia Foundation. Retrieved from https://en.wikipedia.org/wiki/variance_reduction

Vujic, J. L., & Martin, W. R. (1991). Vectorization and parallelization of a production reactor assembly code. *Progress in Nuclear Energy*, *26*(3), 147–162.

Wagner, J. C., et al. (2011). Hybrid and parallel domain-decomposition methods development to enable monte carlo for reactor analyses. *Progress in nuclear science and technology*, *2*(1), 815–820.

Wang, Y., Brun, E., Malvagi, F., & Calvin, C. (2016). Competing energy lookup algorithms in monte carlo neutron transport calculations and their optimization on cpu and intel mic architectures. *Procedia Computer Science*, *80*, 484–495.

Wang, Y., Qin, Q., SEE, S. C. W., & Lin, J. (2013). Performance portability evaluation for openacc on intel knights corner and nvidia kepler. *HPC China*.

What is gpu computing? (2015). NVIDIA. Retrieved from `http://www.nvidia.com/object/what-is-gpu-computing.html`

Wienke, S., Springer, P., Terboven, C., & an Mey, D. (2012). Openacc–first experiences with real-world applications. , 859–870.

Wolfe, M. (2014, jul). *Compilers and more: Mpi+x.* HPC wire. (https://www.hpcwire.com/2014/07/16/compilers-mpix/)

Wolfe, M. (2016, apr). *Compilers and more: What makes performance portable?* HPC wire. (https://www.hpcwire.com/2016/04/19/compilers-makes-performance-portable/)

Xiao, K., Chen, D. Z., Hu, X. S., & Zhou, B. (2015). Monte carlo based ray tracing in cpu-gpu heterogeneous systems and applications in radiation therapy. , 247–258.

Xu, X. G., Liu, T., Su, L., Du, X., Riblett, M., Ji, W., ... others (2015). Archer, a new monte carlo software tool for emerging heterogeneous computing environments. *Annals of Nuclear Energy*, *82*, 2–9.

Yang, F., Yu, G., & Wang, K. (2015). Hybrid shared memory/message passing parallel algorithm in reactor monte carlo code rmc. , 16-18.

Yee, B. C., Olivier, S. S., Southworth, B. S., Holec, M., & Haut, T. S. (accepted (2021), October 3-7). A new scheme for solving high-order dg discretizations of thermal radiative transfer using the variable eddington factor method. *International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering (M&C 2021).*

Yepes, P. P., Mirkovic, D., & Taddei, P. J. (2010). A gpu implementation of a track-repeating algorithm for proton radiotherapy dose calculations. *Physics in medicine and biology*, *55*(23), 7107.